USING VHDL-AMS FOR ELECTRICAL, ELECTROMECHANICAL, POWER ELECTRONIC AND DSP-ALGORITHM SIMULATIONS

P J Randewijk and H du T Mouton

University of Stellenbosch, Dept. of Electrical and Electronic Engineering, Stellenbosch, South Africa

Abstract. This paper will discuss the applicability of VHDL-AMS for multi-domain system - and complex power electronic topology simulations.

Key Words. VHDL-AMS, Simplorer, Modelling, Simulation, Multi-domain, Multi-physics, Power Electronics

1. HISTORY OF VHDL-AMS

VHDL was originally developed by the US Department of Defense to simulate the behaviour of ASIC devices. The US DoD granted all language definition rights to the IEEE and so the IEEE Standard 1076-1987 (IEEE Standard VHDL Language Reference Manual) was born. This sparked industrial interest and extensive investment in VHDL. Although initially intended as a simulation language only, VHDL quickly developed into a language not only capable to simulate and design ASICs, but also to implement complex logic designs into programmable logic devices (e.g. CPLD and FPGA type devices). Some companies, e.g. Altera and Xilinx, provide free VHDL compilers to program their products with. The VHDL language have since been revised twice. The current standard is called the IEEE Standard 1076-2002.

In the early 1990s the need to simulate systems consisting of a combination of analogue and digital signals lead to the establishment of the IEEE Working Group 1076.1. Working Group 1076.1 developed a set of extensions to VHDL language that enabled the simulation of analogue and mixed signal systems. These extensions were published as the IEEE Standard 1076.1-1999, (IEEE standard VHDL analog and mixed-signal extensions). IEEE Standard 1076.1 was coined VHDL-AMS by industry to distinguish it from the original VHDL.

Due to VHDL-AMS's powerful extended instruction set, industry was (and still is some what) reluctant to try and develop analog and mixed signal hardware that could be programmed using the full extend of VHDL-AMS as this seems to be an almost impossible task. VHDL-AMS was deemed more applicable to the simulation of mixed-signal systems (i.e. containing both analogue and digital elements, and the boundary between them) with the focus on IC design.

A number of companies therefore started to develop simulation software based on VHDL-AMS for this purpose, e.g. Ansoft and MentorGraphics.

The strength of VHDL-AMS however became apparent in the multi-disciplinary modelling and design of electro-mechanical, mechatronic and micro electro-mechanical systems (MEMS).

To cater for multi-disciplinary modelling, a further set of extensions to VHDL-AMS was proposed in 2001 and was finally adopted by the IEEE in 2004, as the IEEE Standard 1076.1.1-2004 (IEEE Standard VHDL Analog and Mixed-Signal Extensions-Packages for Multiple Energy Domain Support). These extensions defined a standard set of *terminals*, *natures*, *quantities*, *units*, *symbols*, *constants* and *tolerances* for a common standard interface across the various domains that could co-exist within a "*multiple energy domain*" VHDL-AMS simulation.

2. A BRIEF INTRODUCTION TO VHDL-AMS

2.1 Some of the new keywords

In order to simulate the analogue behaviour of an analogue `entity`, certain extensions to VHDL's external view (i.e. ports) were introduced. For a digital `entity`, all ports were regarded as signals [1]. For an analogue `entity`, ports can now also be declared as *terminals* or *quantities* [2].

The following additional key-words were therefore introduced:

- `terminal`
- `nature`
  - `through`
  - `across`
- `quantity`

2.2 Our first example

In order to demonstrate this, let us consider an example of a simple resistor `entity` in VHDL-AMS, as shown in listing 1.

In line 1, it tells the VHDL-AMS simulator that we want to use the `IEEE library` as defined by the IEEE Standard 1076.1.1-2004, and in line 3 that from this `library` we want to `use all` of the definitions as defined in the `electrical_system` package.

```
1  library ieee;
2
3  use ieee.electrical_systems.all;
4
5  entity resistor is
6    generic(
7      R : resistance := 1.0 -- [Ohm]
8          );
9    port(
10     terminal t1, t2 : electrical
11       );
12 end entity resistor;
```

Listing 1: A VHDL-AMS "interface description" for a resistor.

```
14 architecture simple of resistor is
15   quantity v across t1 to t2;
16   quantity i through t1 to t2;
17 begin
18   v==i*R;
19 end architecture simple;
```

Listing 2: A VHDL-AMS "behavioural model" for a resistor.

Starting in line 5, we describe how the resistor's `entity` (interface description) should look like.

First we need to list all of the `entity`'s properties or parameter. Our simple resistor has only one `generic` parameter, its resistance value, with `resistance` a `subtype` of `real` as defined in the `electrical_system` `package`. We can assign a default value for each `generic` parameter, if required. For our resistor, the default value was chosen as $1\,\Omega$, line 7.

Secondly, in line 10, we describe how our our resistor can be connected to other entities. We define our simple resistor as a two `terminal` device, and that these *terminals* are `electrical` by `nature` as defined in the `electrical_system` `package`. This permits us to connect our resistor to other *terminals* that is also of the `electrical nature` type.

The next step is to define the behavioural model of our resistor. See listing 2.

Here, the first step is to define the `across` and the `through` *branch quantities* associated with the `electrical` *terminals*. This is done in lines 15 and 16. For the `electrical_system` `package`, the predefined `across type` is `voltage` and the predefined `through type` is `current`, both also of the `subtype real`.

Finally, we are in the position to write down the "differential equations" for our simple resistor model that the VHDL-AMS analogue solver will "try" and solve for each analogue time step. This is done with a *simultaneous statement* ('`==`' notation). For our simple resistor, it is a mere one line of code (line 18) which is nothing else but "Ohm's Law", (1).

$$v_R(t) = i_R(t) \cdot R \tag{1}$$

What the *simultaneous statement* does, is that it instructs the analogue solver (e.g. Simplorer®) to find solution at each analogue time step for the *quantities* on both sides of the '`==`' so that they are equal.

2.3 Our second example

Let us consider a more complex example, and that of an inductor, see Listing 3.

For our inductor's `entity` description, two `generic` parameters were chosen, the inductance '`L`' (line 7) and the parasitic series resistance, '`rL`' (line 8). Both were also supplied with a default value.

```
1  library ieee;
2
3  use ieee.electrical_systems.all;
4
5  entity inductor is
6    generic(
7      L  : inductance := 1.0E-3; -- [H]
8      rL : resistance := 1.0E-3  -- [Ohm]
9           );
10   port(
11     terminal t1, t2 : electrical
12       );
13 end entity inductor;
14
15 architecture simple of inductor is
16   quantity v across i through t1 to t2;
17 begin
18   v==L*i'dot;
19 end architecture simple;
20
21 architecture more_complex of inductor is
22   quantity vt across iL through t1 to t2;
23   quantity vL : voltage;
24 begin
25   vL==vt-rL*iL;
26   vL==L*iL'dot;
27 end architecture more_complex;
```

Listing 3: A VHDL-AMS inductor model.

For the behavioural model, we decided that we want to have a '`simple`' model for our inductor that ignores the parasitic series resistance, line 15–19, as well as a '`more_complex`' model that takes the parasitic series resistance into account, line 21–27.

The `through` and `across` *branch quantities* for our '`simpel`' inductor model are defined in a concise manner on line 16 as appose to the "two line" declaration used for our resistor model.

A new attribute, the '`dot`' or *derivative of time* attribute is introduced in line 18 in order to write the "differential equation" for our simple inductor,

$$v_L(t) = L\frac{d}{dt}i_L(t)$$

as a *simultaneous* VHDL-AMS *statement*.

An alternative approach could have been to write the differential equation in integral format, using the '`integ`' attribute and replacing line 18 with,

```
...
i==v'integ/L;
...
```

to realise,

$$i_L(t) = \frac{1}{L}\int_0^t v_L(t)$$

For the '`more_complex`' simulation model of our inductor, the parasitic resistor value, is taken into account. Once again the `across` and `through` *branch quantities* needs to be defined, line 22. The names chosen for the *branch quantities* need not be the same as for the '`simple`' model, as the scope of the names will only be limited to the architectural model in which it was defined.

In line 23 an additional *free quantity* is defined. The allows (forces) us to define another *simultaneous*

Table 1: Multi domain natures available in VHDL-AMS

nature	across	through
electrical	voltage	current
magnetic	mmf	flux
translational	displacement	force
translational_velocity	velocity	force
rotational	angle	torque
rotational_velocity	angular_velocity	torque
fluidic	pressure	vflow
thermal	temperature	heat_flow

statement to cater for the voltage drop across the parasitic series resistor, see line 25, so that the *simultaneous statement* defined in line 26 is virtually the same as that used for the 'simple' inductor model, line 18.

Because our inductor now has two architectures, we can choose which one to use in our simulation, as long as the entity description is *generic* to both.

3. MULTI DOMAIN MODELLING

3.1 Introduction

So far only *quantities* of the electrical nature has been presented.

In this section an example of a simulation with a electrical and a magnetic nature will be shown, followed by a simulation with a electrical and a rotational_velocity nature. All the different *natures* that can be used to describe a terminal(s) in VHLD-AMS together with their corresponding across and through *branch quantities*, as defined by the IEEE Standard 1076.1.1, are listed in Table 1.

3.2 An electromagnetic VHDL-AMS example

A transformer winding is a good example of an entity with two types of *natures*, an electrical nature, with *branch quantities* of voltage across and current through as well as a magnetic nature, with *branch quantities* of mmf across and flux through. A VHDL-AMS example of a transformer core winding is shown in Listing 4 in order to describing how both of these *natures* can co-exist within a single VHDL-AMS model.

The generic parameters for our winding model is the number of turns, 'N', and the parasitic resistance of the winding, 'r'. This is similar to that of the inductor discussed previously. The major difference between the winding model and that of the inductor, is that the winding has two sets of *terminals*, a set of electrical and a set of magnetic *terminals*, line 17 and 18 respectively.

In the architectural description of the winding, names for the *branch quantities* associated with each terminal's across and through *quantity* are assigned. For each set of branch quantities, a *simultaneous statement* (i.e. "differential equation")

```
1  library ieee;
2
3  use ieee.electrical_systems.all;
4
5  entity winding is
6    generic(
7      N : in real := 1.0;       -- [turns]
8      r : in resistance := 0.0 -- [Ohm]
9           );
10   port(
11     terminal e1, e2 : electrical;
12     terminal m1, m2 : magnetic
13       );
14 end entity winding;
15
16 architecture simple_linear of winding is
17   quantity vt across i through e1 to e2;
18   quantity f  across phi through m1 to m2;
19 begin
20   vt==-N*phi'dot+i*r;
21   f ==N*i;
22 end architecture simple_linear;
```

Listing 4: The entity description of a simple winding.

is required to describe the behaviour of the winding. The two "differential equations" necessary to model the behaviour of the winding, are shown in (2) and (3), with the VHDL-AMS implementation given in lines 20 and 21 respectively.

$$v_t(t) = -N\frac{d}{dt}\Phi + iR \tag{2}$$

$$\mathcal{F} = Ni \tag{3}$$

Because VHDL-AMS is a strongly typed language, only *terminals* of the same nature can be connected together.

To illustrate this, an Ansoft Simplorer® SV (Student Version) [3] schematic of a push-pull swith-mode power supply is shown in Figure 1. In the schematic it can be seen that the circuit to the left of windings **n01** and **n01** and that to the right of windings **n011** and **n012** are electrical circuits. Where as in between the winding, a magnetic circuit exists. Simplorer® automatically assigns different colours to different "types" of circuits. The defaults are 'black' for electrical circuits and 'orange' for magnetic circuits.

Also shown (for interest sake), is Simplorer®'s built-in Jiles-Atherton model for a 3F3 Ferroxcube ferrite core. The only parameter that needs to be supplied are the effective lenght (l_e) and the effective area (A_e). For an EFD10 core, $l_e = 23{,}1\,\mathrm{mm}$ and $A_e = 7{,}2\,\mathrm{mm}^2$. For this Simplorer® simulation, the core is split in two, and a leakage reluctance, **Rleak**, is added to simulate the effect of non ideal coupling between the primary and secondary due to (say) the glass tape used for isolation purposes.

Simplorer® has buit-in models for a large range of ferrite cores for the following manufacturers, AVX, Epcos, Ferroxcube, Magnetics, Micrometals, Steward and TDK.

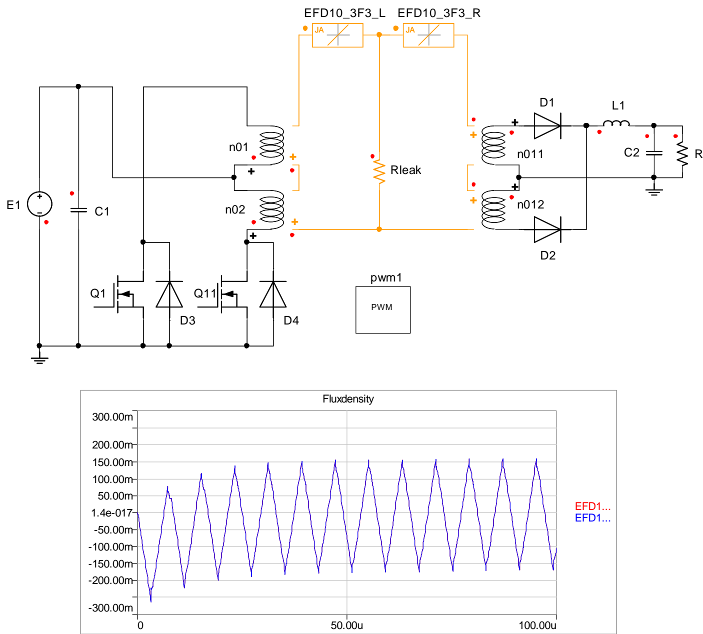

Fig. 1: Simplorer Push-Pull Converter Simulation

3.3 An electromechanical VHDL-AMS example

An permanent magnet DC motor is a good example of a typical electromechanical system [4], that has both an `electrical` and a "mechanical" `nature`. A VHDL-AMS model for the electromechanical behaviour of a permanent magnet DC motor is shown in Listing 5.

Once again the `generic` parameters are listed in the `entity` declaration (line 9–18). For the permanent magnet DC motor, they are:

- armature resistance ('R_a')
- armature inductance ('L_a')
- motor torque constant ('K_T')
- armature moment of inertia ('J_a')
- armature damping ('B_a')

The next step is to define the set of `port` definitions for our VHDL-AMS model. The first `port` definition, line 22, are the (now familiar) `electrical` *terminals*. For the second set of *terminals* however, we have a choice between `rotational` or `rotation_velocity` *terminals*.

For servo applications, the `rotational nature` would be a good choice with `angle` and `torque` as the respective `across` - and `through` - *branch quantities*. For a speed control application however, the `rotational_velocity nature` would be a better choice with `rotational_velocity` and `torque` as the respective `across` and `through` *branch quantities*. Our example choose the latter, see line 24.

What is interesting to note, is that only one `terminal` 'm' was specified. We will come to that later.

Another interesting thing to note, is the declaration in line 26. Here a new type of `port` is defined, a *port quantity*, 'n'. This *port quantity* (the speed of the motor in rpm) is also defined as a ***out*** *quantity*. *Port quantities* can either be declared as **in** or **out** (as appose to a *signal* that can be **in**, **out** or **inout**). This *port quantity* can now be used for plotting - or control purposes (see the next section).

```
 1 library ieee;
 2
 3 use ieee.math_real.all;
 4 use ieee.electrical_systems.all;
 5 use ieee.mechanical_systems.all;
 6
 7 entity pmdcm is
 8   generic(
 9     -- Armature Resistance
10     R_a : resistance := 1.0;
11     -- Armature Inductance
12     L_a : inductance := 1.0E-3;
13     -- Motor Torque Constant
14     K_T : real := 1.0;
15     -- Armature's Moment of Inertia
16     J_a : moment_inertia := 1.0E3;
17     -- Armature's Damping
18     B_a : damping := 1.0E3
19         );
20   port(
21     -- Electrical Terminals
22     terminal t1, t2   : electrical;
23     -- Rotational Velocity Terminal(s)
24     terminal m : rotational_v;
25     -- Speed of the Motor in rpm
26     quantity n : out real
27       );
28 end entity pmdcm;
29
30 architecture simple_linear of pmdcm is
31   constant o2n : real := 60.0/(2.0*math_pi);
32   quantity v_t across i_a through t1 to t2;
33   quantity omega_m across tau_m through m to
34     rotational_v_ref;
35 begin
36   v_t  ==i_a*R_a+L_a*i_a'dot+K_T*omega_m;
37   tau_m==K_T*i_a-J_a*omega_m'dot-B_a*omega_m;
38   n    ==o2n*omega_m;
39 end architecture simple_linear;
```

Listing 5: An Electromechanical VHDL-AMS model of a Permanent Magnet DC Motor.

In the course of this paper, we have now discussed all types of quantities defined in VHDL-AMS:

- *branch quantities*
- *free quantities*
- *port quantities*

In the `architecture` declaration, `across` and `through` *branch quantities* are once again assigned for each set of *terminals*. For the `electrical` *terminals*, this is not a problem. But what about our "single" `rotational_velocity terminal`?

When we measure the speed of a motor's shaft, we usually measure it with respect to standstill. This is similar when measuring a line voltage with respect to ground potential. In VHDL-AMS a reference (i.e. ground) is defined for each type of `nature`. For `rotational_velocity` this is the `rotational_velocity_ref` and for `electrical` it is the `electrical_ref`. Thus we can define our *branch quantities* as shown in line 33 and 34 were `rotational_v_ref` is an `alias` for `rotational_velocity_ref`.

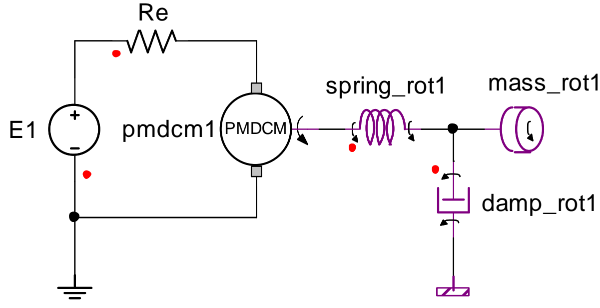

Fig. 2: Electromechanical system using a VHDL-AMS model of an permanent magnet DC machine.

For each set of *branch quantities* a *simultaneous statement* is once again required, as well as for the *port quantity* defined. The differential equations necessary to describe the `electrical` and `rotational_velocity nature` of the permanent magnet DC motor is given in (4) and (5) and implemented in lines 36 and 37 respectively.

$$\begin{aligned} v_t &= i_a R_a + L\frac{d}{dt} i_a + e_a \\ &= i_a R_a + L\frac{d}{dt} i_a + K_T \omega_m \end{aligned} \tag{4}$$

$$\begin{aligned} \tau_m &= \tau_e - J_a \frac{d}{dt} \omega_m - B_a \omega_m \\ &= K_T i_a - J_a \frac{d}{dt} \omega_m - B_a \omega_m \end{aligned} \tag{5}$$

The *simultaneous statement* to convert the speed of motor for rad/s to rpm for *port quantity* '`n`', is shown in line 38.

This VHDL-AMS model was imported into a custom library inside Simplorer® SV to create a new VHDL-AMS element (i.e component). A custom symbol for the new element was created and is shown in Figure 2 together with other predefined "electrical" and "mechanical" components to simulate a coupled electromechanical systems.

The figure clearly shows how we can connect a `rotational_velocity` "circuit" to our DC motor to simulate the behaviour of a mechanical load...

4. CONTROL SYSTEM MODELLING IN VHDL-AMS

The examples shown up till now, focused only on the modelling of an existing "plant", but did not address the modelling of a control system to be used. Analogous to the *derivative of time*, '`dot`' attribute, VHDL-AMS provides two attribute ideally suited to simulate transfer functions in either the continuous time domain or the discrete time domain, by either performing a Laplace transform (using the `ltf` attribute) or a Z-transform (using the `ztf` attribute) on a *port quantity*.

It it therefore possible to simulate a typical lead-lag compensator (6), in VHDL-AMS as shown in Listing 6.

```
entity lead_lag is
  generic(
    K   : real := 1.0; -- Gain
    z_1 : real := 1.0; -- 1st zero
    z_2 : real := 1.0; -- 2nd zero
    p_1 : real := 1.0; -- 1st pole
    p_2 : real := 1.0  -- 2nd pole
       );
  port(
        quantity input  : in real;
        quantity output : out real
         );
end entity lead_lag;

architecture implementation of lead_lag is
  constant num : real_vector :=
    (K*z_1*z_2, K*(z_1+z_2), K);
  constant den : real_vector :=
    (p_1*p_2, p_1+p_2, 1.0);
begin
  output==input'ltf(num,den);
end architecture implementation;
```

Listing 6: A VHDL-AMS model for a Lead-Lag Compensator, in the Continuous Time Domain

$$\begin{aligned} G_c(s) &= K\frac{(s+z_1)(s+z_2)}{(s+p_1)(s+p_2)} \\ &= K\frac{s^2+(z_1+z_2)s+z_1 z_2}{s^2+(p_1+p_2)+p_1 p_2} \end{aligned} \tag{6}$$

On thing to note however, is that the '`num`' and '`den`' constants used for the numerator and denominator coefficients of the transfer function, must be in ascending powers of s.

It is now possible to once again import our Lead-Lag compensator as a "block" into Simplorer® and use it together with Simplorer®'s own "blocks" for "hybrid" control purposes, as shown in Figure 3.

To simulate a discrete transfer function, the '`ztf`', Z-Domain Transform, attribute is used analogous to the '`ltf`' attribute, but with an additional sampling time parameter that also needs to be supplied.

5. DSP ALGORITHM SIMULATIONS IN VHDL-AMS

Most of the examples discussed so far could just as easily have been simulated in Simplorer® without using any VHDL-AMS models, using the predefined circuit- or block elements instead.

However, when trying to simulate somewhat more complex power electronic systems, e.g. space vector pulse width modulation (SVPWM) for voltage source inverter (VSI), or active power filters (APF), or multilevel converters or matrix converter, block element does not suffice to simulate the control

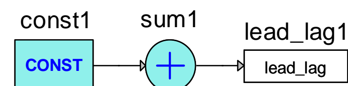

Fig. 3: Hybrid control system using VHDL-AMS blocks together with predefined Simplorer blocks.

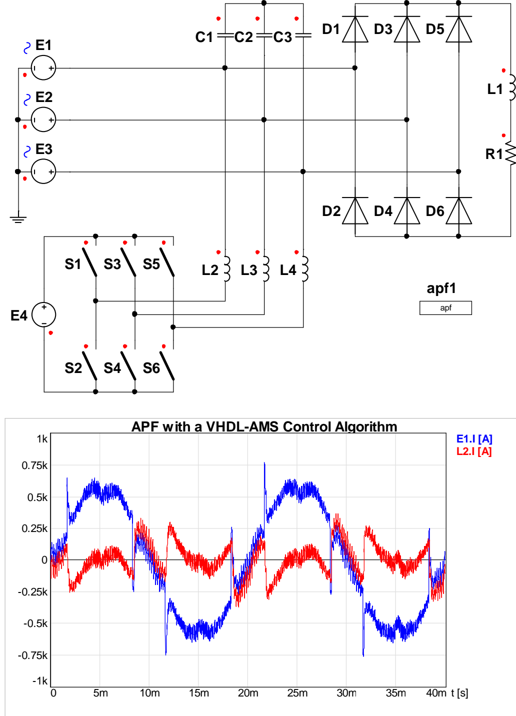

Fig. 4: An Active Power Filter utilising Space-Vector Pulse Width Modulation with all the controls implemented in VHDL-AMS.

system. Simplorer® does have the ability to simulate these type of control systems using graphical state machines, but these state machines simulations are difficult to debug and not easily portable. Also these graphical state machine simulation differs completely from the DSP and/or FPGA code that will be used to implement these control algorithms.

This is where VHDL-AMS really comes to the fore. With the standard 'if-then-else' or the 'case-when' conditional statements, the 'for-loop' loop statement, the ability to define custom *functions* and *procedures* and the IEEE's math_real package (i.e. for 'sin', 'cos', 'tan', 'sqrt', etc. mathematical functions), Simplorer® can be used to simulate C-like code with the minimum amount of porting, using VHDL-AMS.

This will be illustrated using a Simplorer® SV simulation of an APF for a three-phase six-pulse rectifier that utilises SVPWM. The Simplorer® schematic is shown in Figure 4.

Due to the limited space available, only parts of the VHDL-AMS code will be presented however. The "main" section of the APF's control algorithm is shown in Listing 7.

In order to calculate the reference voltage for the VSI, the supply voltage, the supply current and the load current needs to be "measured" by our VHDL-AMS control block. For each value that needs to be "measured" a *port quantity* needs to be defined in our VHDL-AMS model, e.g. v_s_a, for phase *a* of the supply voltage, etc.

As soon as an instance of our VHDL-AMS model

```
83  -- Calculate new duty cycle value every 1/(fs*2)
84  if update = clk_hi_val then
85    v_s_abc:=(v_s_a, v_s_b, v_s_c);
86    v_s_zab:=abc_to_zab(v_s_abc);
87
88    i_s_abc:=(i_s_a, i_s_b, i_s_c);
89    i_s_zab:=abc_to_zab(i_s_abc);
90
91    i_l_abc:=(i_l_a, i_l_b, i_l_c);
92    i_l_zab:=abc_to_zab(i_l_abc);
93
94    p_l     :=dotp(v_s_zab,i_l_zab);
95
96    g_l     :=p_l/(v_s_zab(zero) **2+
97                   v_s_zab(alpha)**2+
98                   v_s_zab(beta) **2);
99
100   i_c_zab:=(i_s_zab(zero) -g_l*v_s_zab(zero),
101             i_s_zab(alpha)-g_l*v_s_zab(alpha),
102             i_s_zab(beta) -g_l*v_s_zab(beta));
103
104   v_r_zab:=(L_f*fs*i_c_zab(zero) +v_s_zab(zero),
105             L_f*fs*i_c_zab(alpha)+v_s_zab(alpha),
106             L_f*fs*i_c_zab(beta) +v_s_zab(beta));
107
108   D       :=space_vector_dcc(v_r_zab,V_dc);
109 end if;
```

Listing 7: The "main" section of the APF's Control Algorithm.

is used, (**apf1** in this case), the *output name* of the *output property* of the Simplorer® *element* that is required to be "measured" must to be typed next to the appropriate *port quantity* of **apf1**.

E.g. with **E1** the AC voltage source used for phase *a*, **E1.EMF** needs to be typed next to the v_s_a parameter name in the **apf1** block.

In order to calculate the voltage reference for the VSI in space vector format, the supply voltage, supply current and load current needs to be converted to space vector format. This is done in lines 85 – 92 by making use of a custom VHDL-AMS function abc_to_zab.

The instantaneous power delivered to the load can now be calculated using the dot product of the supply voltage and the load current [5] as shown in (7) and implemented in line 94 using a custom VHDL-AMS function dotp.

$$p_l(t) = \mathbf{v}_s(t) \cdot \mathbf{i}_l(t) \tag{7}$$

The next step is to calculate the instantaneous conductance of the load, as shown in (8) and implemented in lines 96 – 98.

$$g_l(t) = \frac{p_l(t)}{||\mathbf{v}_s(t)||^2} \tag{8}$$

The compensation current that needs to be injected by the APF, can now be calculated as shown in (9) and is implemented in lines 100 – 102.

$$\mathbf{i}_c(t) = \mathbf{i}_l(t) - g_l \mathbf{v}_s(t) \tag{9}$$

With the switching frequency (f_s) and the filter inductance (L_f=**L2.L**=**L3.L**=**L4.L**) known, the reference voltage for the VSI can be calculated as shown in (10) and implemented in lines 104 – 106.

```
164 -- Space Vector Duty Cycle Calculation --
165 function space_vector_dcc
166     (v_ref_zab : in space_vector;
167      Ud        : in real)
168      return real_vector is
169
170   variable v_ref_sec : real;
171   variable d0, d1, d2, d3, d4, d5, d6 : real;
172   variable DA, DB, DC : real;
173
174 begin
175   v_ref_sec := sec_det_06(v_ref_zab);
176
177   case (v_ref_sec) is
178
179     when 1.0=>d1:=(v_ref_zab(alpha)-
180                   v_ref_zab(beta)/tan60)/(C*Ud);
181                d2:=v_ref_zab(beta)/(sin60*C*Ud);
182                d0:=d1+d2;
183                if d0 > 1.0 then
184                  d1:=d1/d0;
185                  d2:=d2/d0;
186                end if;
187                d0:=1.0-d1-d2;
188                DA:=d1+d2+d0/2.0;
189                DB:=d2+d0/2.0;
190                  DC:=d0/2.0;
191 ...
```

Listing 8: Space Vector Duty Cycle Calculation function.

$$\mathbf{v}_r(t) = L_f \frac{\mathbf{i}_c(t)}{\frac{1}{f_s}} + \mathbf{v}_s(t) \tag{10}$$

The duty cycle for the VSI is now calculated using the space_vector_dcc VHDL-AMS custom function, in line 108. A partial listing of the space_vector_dcc function is given in Listing 8. A description of space vector PWM is left to the reader [6] .

As a final example of how DSP-type algorithms can be implemented in VHDL-AMS, the listing of yet another custom VHDL-AMS function, sec_det_06, is given in Listing 9.

This function calculates in which sector the reference voltage is situated, and is called by space_vector_dcc in line 174. This custom VHDL-AMS function together with the abc_to_zab and dotp VHDL-AMS functions, forms part of a VHDL-AMS space_vector_package that was developed by the authors to specifically deal with space vector calculations for complex Simplorer® simulations of advanced power electronic topologies.

6. CONCLUSION

In this paper the most common "extensions" to VHDL that became later became known as VHDL-AMS were presented. The suitability of VHDL-AMS for multi domain system modelling was discussed. Examples of normal electrical, electromagnetic, electromechanical, continuous transfer functions and typical DSP-type control algorithm as used for advanced power electronic modelling was shown.

With the teaching of VHDL to undergraduate engineering students from the first or second year, VHDL-

```
82  -- Sector Detector --
83  function sec_det_06
84      (s : in space_vector)
85      return real is
86
87    variable sector : real;
88
89  begin
90    if s(beta) > 0.0 then
91      if s(beta) > abs(s(alpha))*tan60 then
92        -- Sector II --
93        sector := 2.0;
94      elsif s(alpha) > 0.0 then
95        -- Sector I --
96        sector := 1.0;
97      else
98        -- Sector III --
99        sector := 3.0;
100     end if;
101   elsif s(beta) < -abs(s(alpha))*tan60 then
102     -- Sector V --
103     sector := 5.0;
104   elsif s(alpha) > 0.0 then
105     -- Sector VI --
106     sector := 6.0;
107   else
108     -- Sector IV --
109     sector := 4.0;
110   end if;
111   return sector;
112 end function sec_det_06;
```

Listing 9: A function to calculate in which sector a space vector is situated.

AMS has the potential to be used not just at post graduate level, but also at undergraduate level.

All of the Simplorer® simulations that was presented, was done in the Student Version of Simplorer®, Simplorer® SV further illustrating VHDL-AMS and Simplorer® SV suitability for undergraduate use.

Also, the analogies illustrated by VHDL-AMS, especially when using Simplorer®, between electrical, mechanical, fluidic and thermal systems/circuits may very well redefine how the "Electrotechniques" and "Systems and Signals" type courses can be presented to engineering students in the future, especially when these courses are not only lectured to electrical engineering students, but also to mechanical, mechatronic, industrial, process and civil engineering students.

REFERENCES

[1] P. J. Ashenden, *The Designer's Guide to VHDL*, 2nd ed. Morgan Kaufmann Publishers, 2002.

[2] ——, *The System Designer's Guide to VHDL-AMS*. Morgan Kaufmann Publishers, 2003.

[3] [Online]. Available: http://www.ansoft.com/products/em/simplorer/

[4] C. M. Close, D. H. Frederick, and J. C. Newell, *Modeling and Analysis of Dynamic Systems*, 3rd ed. John Wiley & Sons, Inc., 2002.

[5] F. Z. Peng, J. Ott, G.W., and D. Adams, "Harmonic and reactive power compensation based on the generalized instantaneous reactive power theory for three-phase four-wire systems," *IEEE Trans. Power Electron.*, vol. 13, no. 6, pp. 1174–1181, Nov. 1998.

[6] H. van der Broeck, H.-C. Skudelny, and G. Stanke, "Analysis and realization of a pulsewidth modulator based on voltage space vectors," *IEEE Trans. Ind. Applicat.*, vol. 24, no. 1, pp. 142–150, 1988.