An Open VHDL-AMS Simulation Framework

T. Schneider, J. Mades, M. Glesner
Institute of Microelectronic Systems
Darmstadt University of Technology
Darmstadt, Germany

A. Windisch, W. Ecker
Corporate Development
Infineon Technologies AG
Munich, Germany

Abstract
The increasing importance of mixed-signal design among today's and tomorrow's hardware systems brings up new challenges in the field of design tool construction. We present a newly developed VHDL-AMS simulation framework. It consists of a VHDL-AMS compiler, elaborator, and simulator comprising of a digital kernel and an analog kernel. The latter has an open interface for the integration of different analog solvers. A coupling with a MATLAB kernel is described in this paper. Furthermore, the framework provides an interface for the integration of different synchronization algorithms. The presented framework has an open object-oriented architecture which provides good capabilities for research in the field of mixed-signal simulation.

1 Introduction
The increasing importance and the growing amount of mixed-signal hardware designs gave rise to the development of new mixed-signal hardware description languages (MS-HDLs). In comparison to traditional analog description languages, such as SPICE, the new MS-HDLs allow mixed analog/digital hardware descriptions on different levels of abstraction ranging from transistor to system level. Moreover, standardization of MS-HDLs accelerates development of language specific mixed-language simulators thus providing the foundation for the exchange of mixed-signal hardware models. One of these languages is VHDL-AMS [1] which was standardized by IEEE last year and has found wide acceptance in industry since. Currently, VHDL-AMS is integrated into several industrial design flows and, caused by this activity, research on language specific issues is increasing. Research directions include development of efficient synchronization algorithms for mixed-language simulators as well as development of new solvability checks for hierarchical VHDL-AMS models.

A first approach of a VHDL-AMS simulator was presented in [2]. In [3] a distributed solution is introduced. In addition, several commercial simulators are now available [4, 5, 6]. In this paper, we present a JAVA based VHDL-AMS simulation environment. It is intended as a basis for future research on mixed signal simulation. The next chapters will describe the overall architecture, the simulation data structure and the simulator components. A detailed example will show the results of a simulation run.

2 Overall structure
Our VHDL-AMS simulation framework is based on an already existing VHDL-AMS analysis environment consisting of a VHDL-AMS compiler and a VHDL-AMS elaborator. The environment was implemented in JAVA, which gained us platform independence and powerful class libraries. The compiler translates the VHDL-AMS design files into an intermediate representation. The intermediate is structured as a concrete syntax tree with annotated symbol tables and additional semantic information.

![Design Flow](image)

Figure 1: Design Flow

After compilation the design is elaborated. Thereby, the hierarchical design is flattened to a list of time-discrete(digital) processes and the time continuous(analog) equation sets. The simulation framework
consists of a digital kernel and an analog kernel with an interface to different analog solvers for mathematical calculations. The components of the VHDL-AMS simulator and the data structures used for simulation are described in detail in the following chapters. For a more detailed description of the compiler and the elaborator the interested reader is referred to [7, 8].

3 Simulation data structure

Our simulation intermediate structure is an object oriented data structure implemented in JAVA. It is built by the VHDL-AMS elaborator [8] which evaluates dynamically pre-compiled VHDL-AMS design units [7]. The intermediate represents the hierarchical structure of the model consisting of objects representing language scopes, concurrent statements and simultaneous statements. Objects representing language scopes contain an activation record which holds instances of VHDL-AMS objects, types, and values. These objects are interconnected as shown in figure 3 and therefore represent the complete hierarchical signal flow (digital) as well as the flow of the quantities (analog) and the connection of terminals. For the digital VHDL object signal, signal drivers are annotating explicit and implicit signal instances. Signal drivers store transactions with each transaction denoting a time-value pair of the corresponding signal. In case of explicit signals the signal driver also annotates the VHDL process instance of the hierarchical intermediate structure in which the signal appears as target of a signal assignment. These annotated process instances furthermore store information about objects being read inside its scope.

Figure 2: Elaborated data structure

Elaboration of the analog parts of the model results in the generation of an explicit and an implicit set of characteristic expressions (CEs) that have to be solved by the analog solver. The implicit set is a static set resulting from structural information of the model. It is built during elaboration of branch quantity declarations and during the evaluation of hierarchical interconnection (ports/port map aspects).

The explicit set is the set of explicitly described equations in the model using simultaneous statements provided by VHDL-AMS. Because of control elements in the simultaneous statements (if/else) this set is a dynamic set that has to be re-built at each time the analog solver is executed. During simulation additional augmentation sets are determined.

Characteristic expressions are represented in the following form: An operand interface is implemented by all allocated VHDL-AMS objects and literals. The operators, which implement the operand interface, are divided into three subclasses: binary and unary operators as well as a class for function calls. With these two objects the left associative expression structure is built and stored in a characteristic expression object corresponding to a set object. Figure 3 illustrates the characteristic expression \( V = I \cdot R \) of a resistor model stored in the explicit set.

After this discussion of the simulator internal data structure we now focus on the class hierarchy underlying our VHDL-AMS object implementation. Object oriented languages like JAVA allow class inheritance and polymorphism. Based on these two concepts an efficient object structure containing different classes of VHDL-AMS objects was implemented. Figure 4 shows the partial class hierarchy for VHDL signals.

Figure 3: Characteristic expression of a resistor model

We distinguish between VHDL-AMS objects of different types. Thus, a signal declaration of type integer is represented by a scalar signal instance whereas a signal declaration of an array type results in the creation of an array signal instance which stores n scalar signal instances.

4 Simulator components

The simulator framework consists of a digital kernel, an analog kernel, an interface to different analog
solvess and an open scheduler interface for the implementation of different synchronization algorithms. The complete simulator accesses the above described global data structure. The particular components are discussed in the next sections.

4.1 Digital kernel

The digital kernel contains a set of elementary processes \( \{p_1, \ldots, p_n\} \), which are activated by the scheduler. Such processes are *process statements, concurrent signal assignments, concurrent assertions, concurrent procedure calls and concurrent break statements*. These processes can access the above described elaborated data structure which includes the access to the corresponding parse-trees.

After elaboration all composite signals are flattened to their scalar sub-elements. This set of signals is updated in every simulation cycle. The update mechanism is encapsulated in the above described data structure. Each process is associated with an interpreter that executes the corresponding sensitive process until a *wait statement* occurs. The execution of a wait statement causes a suspend exception to be thrown which contains the new *sensitivity set or time out clause* and the *wait statement node*. This information is stored in the process data structure. In the next simulation cycle, this information is used to determine the resumption and the sensitivity of the process. The kernel calls are invoked by a scheduler, which can be easily adapted to different scheduling algorithms. By default, the digital kernel runs as a VHDL-93 digital simulator. For mixed-signal simulation the kernel is extended by an analog kernel which is described in the next chapter.

![Figure 4: Object Hierarchy of the VHDL object signal](image1)

![Figure 5: The architecture of the simulation framework](image2)

4.2 Analog kernel

The analog kernel is called by the scheduler to calculate the values of the quantities in a time interval current time \( T_i \) and time of the next digital event \( T_n \). It divides this interval into a couple of time-steps \( T_i \) were an analog solution point (ASP) has to be found as described below. Additionally the kernel guards the implicit signals corresponding to the attribute \( Q_{above}(E) \). If the calculated value of the quantity \( Q \) crosses the threshold defined by the expression \( E \), an event occurs on the implicit signal and the analog solvers suspends at its local time \( T_i \).

4.3 Analog solver interface

The analog kernel provides an interface to adapt different analog solvers for calculating the analog solution point at a specific time step \( T_i \). To calculate an ASP the analog kernel has to export all the unknowns of the structural- and the explicit equation sets as well as the unknowns of the current augmentation set (\textit{exportUnknowns()}). Then the analog solver is invoked to export the equations or the corresponding matrices (\textit{exportEquations()}) from the above described data representation into its internal representation, like symbolic variables or matrix representing data-structures. After this step the calculation of the ASP is performed (\textit{getASP()}) and finally the calculated values of the quantities are passed back to the database objects/quantities by calling the update-method \textit{up-}
The interface was tested by a prototype implementation of such an analog solver using MATLAB[9]. The implementation consists of a C-library to couple the MATLAB-engine to the JAVA analog kernel and a newton-raphson iteration algorithm implemented in the MATLAB programming language. The algorithm uses an additional toolbox, the MATLAB symbolic toolbox to access predefined symbolic functions like jacobian(F,x) to calculate the jacobian of the functions F and the unknowns x. Using this approach of symbolic calculations, the performance of the simulation was very weak in difference to a second implementation using numerical calculations. To implement the numerical algorithm the quantity values of previous time-step had to be stored additionally.

Now in time, the Infineon circuit simulator TITAN will be extended towards VHDL-AMS by using the interface to connect to the presented simulation system.

4.4 Scheduler interface
As mentioned above, the open scheduling interface provides an easy adaption of different synchronization algorithms. The scheduler interface provides methods for the integration of sequential and parallel execution of the analog and digital kernel. Therefore the simulation cycle is divided into atomic steps. For each step, the scheduler calls a method at the corresponding analog or digital kernel. In a simple lock-step algorithm, these steps are executed in sequential order. More advanced algorithms can execute some of these steps in parallel. Both, the analog and the digital kernel, work on an uniform data structure. This allows the implementation of efficient synchronization algorithms.

5 Example
The example shows the simulation results for the mixed-signal circuit shown in figure 7 below. This circuit is modeled in structural VHDL-AMS. The nmos transistor (ohmic region) amplifies the sinus input voltage while the schmitt-trigger generates a digital clock impulse comparing the amplified voltage with two threshold voltages.

---

**Example Source Code**

```vhd
entity mae is
end entity;

architecture struct of mae is
terminal drain, gate, vdd: electrical;
signal clk : bit := '0';
begin

NMOS: entity work.mosfet(nmos)
generic map(t:= 1.0,
vth := 0.7)
port map(drain, gate, ground);

R : entity work.res behave
generic map(100.0)
port map(vdd, drain);

VSIN: entity work.vsin behave
generic map (offset := 1.0,
T := 1.0e-3,
mag := 0.1)
port map(v1 := gate,
v2 := ground);

SCHM: entity work.schmitt behave
generic map (v1 := 3.5,
vh := 7.5)
port map (refterm := drain,
s := clk);

VDD1: entity work.vdd behave
generic map (v := 10.0)
port map(drain, ground);
end struct;
```

---
architecture behave of schmitt is
quantity ref: real;
begin
    ref = refTerm 'reference;

    P: process
    begin
        if (ref ' above (vh)) then
            s <= '1';
        else
            if (not (ref ' above (vl))) then
                s <= '0';
            end if;
        end if;
        wait on ref ' above (vh), ref ' above (vl);
    end process;
end behave;

entity voin is
    generic ( offset : real := 0.0; 
              T : real := 1.0; 
              mag : real := 1.0; 
          port ( terminal v1, v2: electrical );
    end entity voin;

architecture behave of voin is
    quantity V across I through v1 to v2;
    begin
        V = offset + 
            mag * sin(2.0 * MATH_PI / T * now);
    end behave;

entity mosfet is
    generic ( tc : real := 1.0; -- K/W/L 
              vth : real := 0.7; -- Vth 
          port ( TERMINAL d, g, s : electrical );
    end entity;

architecture mmos of mosfet is
    quantity vgs across g to s;
    quantity vds across gids through d to s;
    quantity vgd across g to d;
    begin
        if vds >= 0.0 use -- forward
        if (vgs - vth) <= 0.0 use
            ids = 0.0;
        elsif (vgs - vth) <= vds use
            ids = 0.5 + tc *(vgs - vth) + 2
        else
            ids = tc * vds *(vgs - vth) - 0.5*vds
        end use;
        else
            -- reverse
        end use;
    end mmos;

architecture a_res of res is
    quantity Vr across Ir through pr to mr;
    begin
        Vr = R * Ir;
    end;

Our top level architecture of the example circuit is a structural description connecting the single models (VSIN, NMOS, VDD1, SCHM, R) to build the circuit shown in schematic 7. The sinus voltage source VSIN is a behavioral description providing a sinus voltage with the parameters DC-Offset, magnitude and period time. In our example VSIN is used to stimulate the gate of the transistor model around the operating point. The NMOS-transistor model NMOS is a behavioral description describing the different behavior of the drain-source current ids depending on the values of the voltages vgs, vds and vt (threshold voltage). In our example the transistor is working in the ohmic region to amplify the sinus gate-source voltage. In a separate simulation run we simulate the transistor model with an increasing gate-source voltage vgs (see Figure 8).

The model SCHM of the schmitt-trigger is described as a digital process triggered on the implicit signals ref ' above (vh) and ref ' above (vl) of the reference quantity ref of the connected terminal refTerm. In case of the value of the reference quantity being above vh the digital output signal s is high ('1') and changes it's value to low ('0') when the value of the reference quantity is below vl. In our example the connected terminal is the drain of the mosfet-transistor. This causes that the reference quantity compared to the voltages vl and vh equals the drain-source voltage vds. The models R and VDD are simple behavioral models describing the behavior of the linear resistor and the constant source voltage vdd.

Figure 9 illustrates the simulation results of the simulation run.

Figure 8: Drain-Source voltage VDS over Gate-Source voltage VGS
6 Out look

Future research will focus on development of different synchronization algorithms and on checks for solvability of hierarchical VHDL-AMS models. Furthermore, effort will be put into the integration of the Infineon in-house analog simulator TITAN [11].

References


