DEVS for mixed-signal Modeling based on VHDL

Shaylesh Mehta
Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. CANADA.
gwainer@sce.carleton.ca

Abstract. *We show how to build generic DEVS models to facilitate simulation of mixed signal Hardware Description Language models within a DEVS simulator. We present the models required, and the conversion procedures for a subset of VHDL called sAMS-VHLD. Hierarchical models written in sAMS-VHDL that utilize Processes, Signals and Simultaneous Statements are be simulated in the CD++ toolkit by elaborating the model, and converting the model hierarchy into an equivalent CD++ coupled model. These Process, Signal and Integrator models and their associated conversion procedures were designed and then tested in CD++ using a number of characteristic sAMS-VHDL models.*

Keywords: n-VHDL, DEVS, CD++, G-DEVS, Q-DEVS.

1. INTRODUCTION

Today's technology business climate requires hardware designers be fast; not only when designing new technology, but throughout the design and maintenance cycle [1]. Digital designers have known for some time, that thorough modeling and simulation of designs reduces the number of design and integration errors, eases product maintenance and reduces costs. Design and simulation of digital logic with HDLs (Hardware Description Languages) is a well-proven methodology; digital designers have a rich toolset available for defining and verifying logic before manufacturing. Such robust toolsets for analog and mixed signal design and simulation have yet to be developed, and those currently available have many limitations and do not exhibit the desired performance. A suitable mixed signal simulator would give the designer the ability to optimize, debug, and verify designs with lower simulation costs, lower risk on manufacturing investment and faster turnaround time.

A key design challenge for a mixed signal simulator is to provide high performance while maintaining the accuracy of continuous time signals. We also need to provide concurrent execution of the simulations of the digital and analog models. A proposed solution to these challenges presented in [2], is to simulate mixed signal HDL models in DEVS (Discrete EVent Systems Specification) [3]. In order to simulate a mixed signal HDL model within a DEVS simulator, generic models that capture the semantics of the constructs within the HDL must be developed within the DEVS simulator using the developed generic models. We present a proposal for a subset of VHDL-AMS (**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage - **A**nalog **M**ixed **S**ignal) [4], which permits defining digital and analog components. We designed a set of generic DEVS models and presented conversion procedures to simulate designs utilizing these constructs in the CD++ toolset [5], which implements the DEVS formalism. This set of VHDL constructs [6] with analog extensions will be referred to as **sAMS-VHDL** (simple Analog Mixed Signal VHDL).

DEVS theory evolved and it was recently upgraded in order to permit modeling of continuous and hybrid systems [7]. This presents some advantages, including greater accuracy in modeling continuous systems and the ability to develop a uniform approach to model hybrid systems, i.e. composed of both continuous and discrete components. The idea beyond this method is to provide quantization of the state variables obtaining a discrete event approximation of the continuous system.

In the long term, we want to attack the development of hybrid systems based on the DEVS formalism and its extensions, building libraries to make easy to use components developed on top of DEVS modeling tools. Here, we show how to build a sAMS-VHDL interpreter using DEVS-based models. One of the benefits is that for a given accuracy, the number of transitions can be reduced, decreasing the execution time of simulations. Discrete time models can be simulated under the discrete event paradigm, thus allowing the development of a simulation environment for complex systems, modeled as hybrid systems, where all techniques (continuous time, discrete time, discrete event) merge together.

2. THE DEVS FORMALISM AND CD++

DEVS is a theoretical approach, which allows the definition of hierarchical modular models. DEVS models may be described as a set of communicating atomic or coupled submodels. A real system modeled with DEVS is described

as a composite of submodels, each of them being behavioral (atomic) or structural (coupled). A DEVS atomic model is can be informally described as in Figure 1.

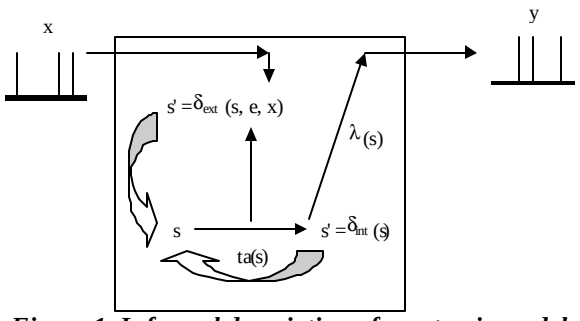

Figure 1. Informal description of an atomic model.

Each atomic model can be seen as having an interface consisting of *input* (x) and *output* (y) *ports* to communicate with other models. Every *state* (**s**) in the model is associated with a *time advance* (**ta**) function, which determines the duration of the state. Once the time assigned to the state is consumed, an internal transition is triggered. At that moment, the model execution results are spread through the model's output ports by activating an *output function* (l). Then, an *internal transition function* (d_{int}) is fired, producing a local state change. Input external events (those events received from other models) are collected in the input ports. An external transition function (d_{ext}) specifies how to react to those inputs.

A DEVS coupled model is composed by several atomic or coupled submodels, as seen in Figure 2.

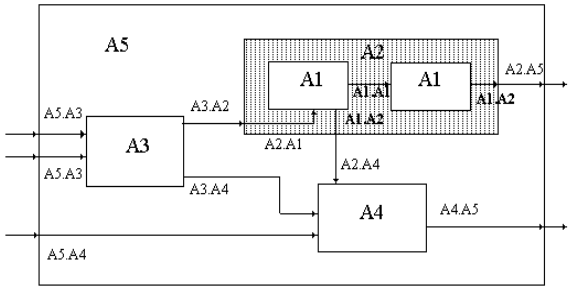

Figure 2. Informal description of a coupled model.

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model's interfaces. The model's coupling defines how to convert the outputs of a model into inputs for the others, and to inputs/outputs to the exterior of the model.

CD++ [5] is a modeling tool that was defined using DEVS specifications. DEVS Atomic models can be programmed and incorporated onto a class hierarchy programmed in C++. Coupled models can be defined using a built-in specification language. CD++ makes use of the independence between modeling and simulation provided by DEVS, and different simulation engines have been defined for the platform: a stand-alone version, a Real-Time simulator, and a Parallel simulator.

CD++ is built as a class hierarchy of models related with simulation processing entities. DEVS Atomic models can be programmed and incorporated onto the *Model* basic class hierarchy using C++.

```
class Atomic : public Model  {
public:
virtual ~Atomic(); // Destructor

protected:
   //Kernel services
   Time nextChange();
   Time lastChange();
   holdIn(AtomicState::State &, Time &);
   passivate();
   ModelState* getCurrentState() ;
   sendOutput(Time &time,Port &port, Value value);

   //User defined functions.
   initFunction();
   externalFunction(ExternalMessage & );
   internalFunction(InternalMessage & );
   outputFunction(CollectMessage & );
   string className() const
   };    // class Atomic
```

Figure 3. The Atomic Class

Once an atomic model is defined, it can be combined with others into a multicomponent model using a specification language specially defined with this purpose. Four properties must be configured: components, output ports, input ports and links between models. The syntax is:

`Components`: `name1[@atomicClass1] name2 ...`

Lists the components of the coupled model (atomic or coupled). For atomic models, an instance and a class name must be specified, allowing a coupled model to use more than one instance of a given atomic class. For coupled models, only the model name must be given.

`Out`: `portname1 portname2 ...`

Enumerates the model's output ports (optional clause).

`In`: `portname1 portname2 ...`

Enumerates the input ports (optional clause).

`Link`: `source[@model] destination[@model]`.

It describes the internal and external coupling scheme. If the name of the model is not included, the default will be the coupled model being defined currently.

3. SAMS VHDL

sAMS VHDL is targeted toward register transfer level modeling of digital circuits with limited behavioral modeling and analog constructs. sAMS VHDL integrates many of the features of VHDL-AMS [4] and explicitly includes some of the types and functions defined by the IEEE 1164 standard logic package [6]. The basic component is the design **entity** declaration, which describes the interface to a sAMS VHDL design unit.

```
entity entity_name is
  { port ( [signal | terminal | quantity]
  identifier{, identifier}: [mode | signal_type
      | electrical]; }+
 end [entity] [entity_name] ;
```

The entity declaration contains a list of ports, each of which is assigned a type and an optional mode. Ports of type *std_logic* or *std_logic_vector* (a standardized type for digital logic) are used for digital signals, while ports of type *electrical* are used for analog signals. In the case of digital signals, ports will have mode *in*, *out*, *inout* or *buffer*. Analog ports do not require a mode.

Figure 4 shows the entity declaration of a digital flip-flop and an analog circuit (low-pass filter). In the flip-flop declaration, *d* and *clk* are input ports of type *std_logic*, and *q* is an output port of type *std_logic*. In addition to the basic std_logic type, vectors of std_logic signals may be declared using the *std_logic_vector* type. This allows digital signals to be operated on by only referencing one signal name. In the declaration for the analog low-pass filter, *tout*, *tin* and *tgnd* are *electrical* ports.

```
entity d_flip_flop is
   port( d, clk : in std_logic;
         q : out std_logic; ) ;
end entity d_flip_flop ;

entity LPF is
  port (terminal tout, tin, tgnd: electrical);
end entity LFP;
```

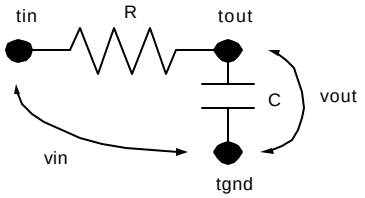

Figure 4. Low-pass Filter

A design **architecture** describes the functionality of a design unit (it may be a structural, dataflow or behavioral description). A single architecture is associated with exactly one entity, whose syntax is:

```
architecture architecture_name of entity_n is
     signal_declaration
     | constant_declaration
     | component_declaration
begin
     {process_statement
     | concurrent_signal_assignment_statement
     | component_instantiation_statement
     | simultaneous_statement}
end [architecture] [architecture_name] ;
```

The body of an architecture is made up of statements that may be categorized as **concurrent**, **sequential** or **simultaneous**. These statements operate on signals/quantities declared within the scope of the architecture, and ports that are declared in the entity the architecture is associated with.

Signals and **quantities** are declared in the declarative region of an architecture. They belong to the scope of the architecture in which they are declared, and may only be referenced within that architecture. Signals and quantities have types (similar to ports in the entities). Types *std_logic* and *std_logic_vector* are used for digital logic. Signals and quantities are defined as below:

```
signal signal_name : std_logic_vector
(upper_bound downto lower_bound) | std_logic ;
quantity identifier: REAL | Voltage | Current |
Charge ;
```

Quantities can also be declared as relative to terminals in an entity, defined as *across* or *through* quantities. Across quantities represent the voltage at the free terminal relative to the reference terminal. Through quantities represent the current from the free terminal into the reference terminal.

```
quantity identifier {, identifier} across iden-
tifier {, identifier} through free_terminal to
reference_terminal ;
```

Concurrent statements within an architecture body execute concurrently. They include statements for *Process*, *Simultaneous*, *Concurrent Assignment* and *Conditional Concurrent Assignments*. The conditional concurrent assignment assigns a target signal using a condition. Instead, the unconditional concurrent assignment always assigns the value of the source signal to the target signal.

```
target_signal <= expression1 when condition
      else expression2; // conditional
target_signal <= source_signal; //unconditional
```

A **process** executes the statements between *begin* and *end process* when an event occurs on a signal in its sensitivity list. All signals modified by the process are updated only when the process body is completed. The statements between *begin* and *end* (sequential statements) are executed in sequence.

```
[process_name:]
process (sensitivity_list) { type_declaration }
begin
  {signal_assignment_statement | if_statement
  | case_statement
end process [process_name] ;
```

The **if-then-else** statement has the same semantic found in most programming languages.

```
[ if_name: ] if  condition  then
                  sequence_of_statements
            {elsif  condition2  then
                  sequence_of_statements  }
            [else sequence_of_statements ]
            end if [ if_name ] ;
```

The **case-when** statement runs the sequence of statements listed under the *when* clause whose expression matches that of the expression in the *case* statement.

```
[ case_name: ] case  expression  is
 {when identifier | expression | discrete_range
    | others => sequence_of_statements}+
endcase [ case_name ] ;
```

The **sequential assignment** assigns the value of the driver signal to the target signal. When executed from within a process, the target will not get the value of the driver until the end of the process.

```
[ label: ] target <= driver ;
```

Simultaneous statements are used for describing Differential Algebraic Equations, and may consist of quantities or signals, including a minimum of one quantity per simultaneous statement (we only support Ordinary Differential Equations). Simultaneous statements may appear anywhere a concurrent statement may, and they have no order.

```
x1'dot'dot == -f*(x1 – x2) / m1;
x2'dot'dot == -f*(x2 – x1) / m2;
```

In the previous example, the *'dot* notation denotes the derivative with respect to time of the quantity. For example, *signal'dot* is the first derivative with respect to the time of the signal, while *signal'dot'dot* is the second derivative.

Components facilitate hierarchical design within sAMS-VHDL. A component instance is a copy of the named entity and its associated architecture that interacts with the architecture it is instantiated within. The port map clause specifies which ports of the entity are connected to which signals in the enclosing architecture body.

```
Instantiation_label :
entity entity_name
port map (
  {port_name => signal_name | expression |
    variable_name | open }+    );
```

4. MAPPING SAMS VHDL TO DEVS

Each of the sAMS-VHDL constructions presented in the previous section must be converted into a DEVS model, and made it available for execution in CD++. **Process** models are translated into CD++ by converting its sequential statements to C++ code, and instantiating ports for every signal that is read or driven from within the process and for every signal in the processes sensitivity list. Figure 5 illustrates an example of a DEVS model generated from a flip-flop.

```
flipflop: process (clk)
begin
    if( rising_edge(clk) )
        q<=d;
    end if;
end process my_proc;
```

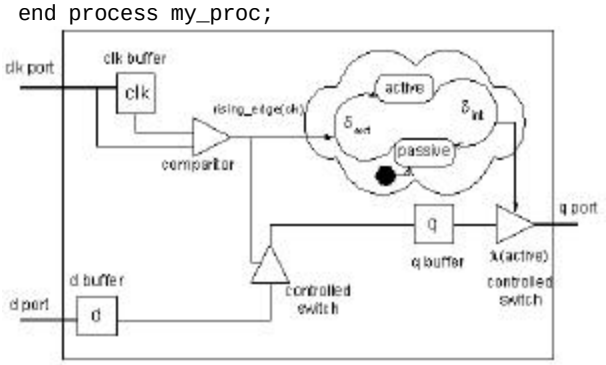

Figure 5. CD++ Process Model.

The process body is implemented within the external transition function. The values received from all external events generated on the input ports (representing read and sensitivity list signals) are buffered within the model. If the process body contains a reference to *rising_edge(signal_name)* or *falling_edge(signal_name)* operations, the values received from the external events are stored on a buffer of length two within the model (keeping the previous and current values of the signal).

The sequential statements in the process body are converted directly to C++ and inserted into the external transition function since they are sequential and semantically equivalent to C++ statements. sAMS VHDL *If*, *case* and *assignment* statements are converted directly into C++ *if*, *switch* and *assignment* statements. The boolean expression that refers to read and sensitivity list signals in the sAMS VHDL if statement is replaced with an equivalent boolean expression that refers to port buffers for those signals.

If the condition within an *if* statement contains a sensitivity list signal, the last piece of code within the C++ *if* condition should instruct the process model to change to the *active* state in 0-time (causing an instantaneous output event and internal transition). The output event will update all driven signals (by sending the value of each output port buffer), while the internal transition will cause the model to return to the *active* state. The following figure shows sAMS VHDL code for a process used in a four bit counter and its translation into CD++.

This process has one sensitivity list signal (*clk*), four read signals (*d1...d4*) and four driven signals (*q1*…q4). The process body contains an *if* sequential statement with a boolean expression that contains the *rising_edge* operation acting on signal *clk*, and four sequential assignment operations.

```
Counter: process (clk) is
begin
     if(rising_edge(clk))
       q1<=not d1;
       q2<=d1 xor d2;
       q3<= d3 xor (d1 and d2);
       q4<=d4 xor (d1 and d2 and d3);
     end if;
end Counter;
```

```
if (msg.port()==clk) {
         // clk is in the trigger list
       o_clk=n_clk;
       n_clk=msg.value();
}
...
//port buffer code for d1 d2 d3 d4
if(o_clk==0 && n_clk==1) {
   // if rising_edge(clk)
   _q1=_1164not(_d1);  q2=_1164xor(_d2,_d1);
   _q3=_1164xor(_d3,_1164and(_d1,_d2));
   _q4=_1164xor(_d4,_1164and(_d3,_1164and
         (_d1,_d2)));
   holdIn(active,0);
}
```

Figure 6. Translating Process Models

We show a fragment of the C++ code generated, in which *o_clk, n_clk, _d1, _d2, _d3* and *_d4* are input port buffers; *_q1, _q2, _q3* and *_q4* are output port buffers, and *_1164and*, *_1164not* and *_1164xor* are functions that implement and, not and xor operators in CD++.

```
signal my_signal, x, y, z: std_logic;
…
my_signal<=x after 5ns;
```

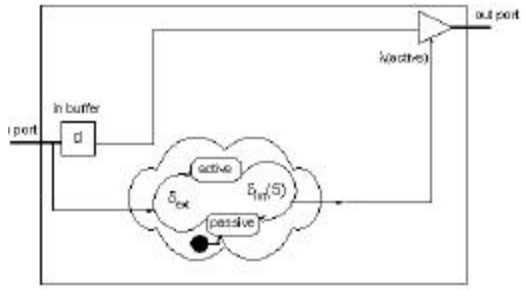

Figure 7. CD++ Signal Model.

Signals are used to determine how to interconnect the ports on the many process model instances for each component. This information is then used during model file generation to create links between the models. As DEVS links provides instantaneous communication between the components, a signal model is created to implement transport delay on messages sent between process model ports. The signal model receives and buffers data on its input port, enters the active state for the time specified by the assignment statement transport delay, then outputs the buffered data on its output port, as showed in Figure 7.

Simultaneous statements in sAMS VHDL allow the definition of ordinary differential equation systems with initial conditions. The problem of simulating an nth ordinary differential equation is solved by reducing the nth order ordinary differential equation into a set of first order differential equations. For example, $\frac{d^2y}{dx^2} + p(x)\frac{dy}{dx} = q(x)$ can be written as two first-order differential equations: $\frac{dy}{dx} = z(x), \frac{dz}{dx} = q(x) - p(x)z(x)$. In general, an nth order ordinary differential equation of form: $F(t, y, y', y'', \ldots, y^{(n)}) = 0$ (1) may be decomposed into a set of first order differential equations:

$\frac{dy_i(t)}{dt} = f_i(t, y_1, \ldots, y_N), \quad i = 1, \ldots, N$ (2) where each $f_i(t, y_1 \ldots y_N)$ is known. A solution for each $y_i(t)$ is obtained for some t >0 and $y_i(0)$ set by integrating each $\frac{dy_i(t)}{dt}$. We have used both Euler's and Fourth-order Runge-Kutta methods (which is more accurate and stable) for the numerical integration [8]. The Runge-Kutta method does not rely only on the derivative at the beginning of the interval only, but also uses the derivative at two trial midpoints and the derivative at a trial end point, as showed in Figure 8.

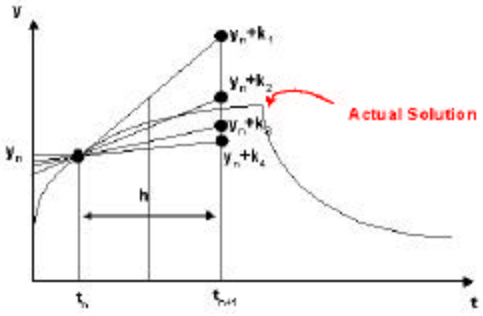

Figure 8. Runge-Kutta Integration

Finally, a weighted sum of k_1, k_2, k_3 and k_4 is added to y_n to determine y_{n+1}.

$$k_1 = hf(t_n, y_n), k_2 = hf(t_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$

$$k_3 = hf(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}), k_4 = hf(t_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \qquad (3)$$

Continuous time ODE systems with initial conditions have traditionally been simulated by discretizing the time domain, and solving the ODE over each discrete time interval. An alternative approach introduced in [7] suggests discretizing the state space of the solution rather than the time domain. Instead of determining what value a dependant variable will have at a given time, we must determine at what time the variable will enter a given state. These systems are termed quantized state systems, and its use may yield results as accurate as a discrete time approach.

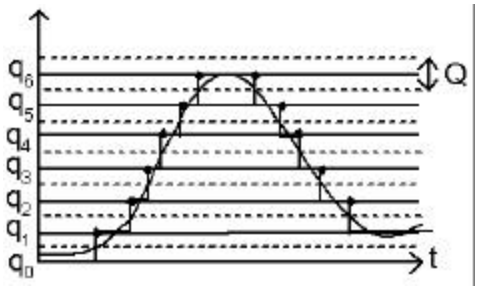

Figure 9. Signal Quantization

In order to use the Fourth-order Runge-Kutta method in a quantized state system, equation (3) must be modified to determine *h* when $y_{n+1} - y_n = Q/2$ (*Q* is the quantum size). Let *Q* be the quantum size. Then, substitute

$k_1 = \frac{Q}{2}, k_2 = \frac{Q}{2}, k_3 = \frac{Q}{2}$ and $k_4 = \frac{Q}{2}$ in (3) to get

$$h_1, h_2, h_3 \text{ and } h_4 h_1 = \frac{\frac{Q}{2}}{f(t_n, y_n)} h_2 = \frac{\frac{Q}{2}}{f(t_n + \frac{h_1}{2}, y_n + sign(h_1)\frac{Q}{4})}$$

$$h_3 = \frac{\frac{Q}{2}}{f(t_n + \frac{h_2}{2}, y_n + sign(h_2)\frac{Q}{4})} h_4 = \frac{Q}{f(t_n + h_3, y_n + sign(h_3)\frac{Q}{2})}$$

Rearrange the sum in (3) and substitute for k_1, k_2, k_3 and k_4

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} y_{n+1} - y_n =$$

$$\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \left| y_{n+1} - y_n \right| = \frac{Q}{2}$$

$$\frac{Q}{2} = \left| h \left[\frac{\frac{Q}{2}}{h_1 6} + \frac{\frac{Q}{2}}{h_2 3} + \frac{\frac{Q}{2}}{h_3 3} + \frac{\frac{Q}{2}}{h_4 6} \right] \right| h = \left| \left[\frac{1}{h_1 6} + \frac{1}{h_2 3} + \frac{1}{h_3 3} + \frac{1}{h_4 6} \right]^{-1} \right| \quad (4)$$

Equation (4) determines at what time relative to the present time the integral of the first order differential equation will enter the quantum state above or below its current quantum state. In order to simulate an ODE system written in sAMS VHDL simultaneous statements, the ODE is first decomposed into a set of first order differential equations. Each of these is then be converted into a Fourth-order Runge-Kutta Quantized Integrator model during Model Code and Netlist Generation, and it is instantiated and linked to other Fourth-order Runge-Kutta Quantized Integrators during the coupled model generation (outlined in Section 5).

The conversion process must first determine which quantities and signals are exogenous and endogenous to the ODE system. Endogenous quantities will be the quantity on the left hand side of the simultaneous statement as well as all quantities on the right hand side of the simultaneous statement with the same quantity name as the left hand side quantity. All other quantities or signals will be exogenous. For example, the following simultaneous statement describes a first order low-pass filter with input voltage *vin* and output voltage *vout*:

```
vout'dot = (1/(R*C))*(vin-vout);
```

In this statement, *vin* is an exogenous quantity, while *vout* and *vout'dot* are endogenous quantities. Once all endogenous and exogenous quantities and signals have been identified, the ODE specified in the simultaneous statement must be decomposed into a set of first order differential equations as outlined in (2). Each of these first order differential equations is then converted directly into a Fourth-order Runge-Kutta Quantized Integrator. Each Integrator must have an input port for each exogenous and endogenous quantity or signal on the right hand side of its first order differential equation, and an output port for the integral of the left hand side of its first order differential equation. For example, the low-pass filter above requires only a single integrator, and this integrator has input ports for vin and vout, as well as an output port for vout.

Following all port buffer code in the integrators external transition function, the model executes the Fourth-order Runge-Kutta method for a quantized state system if the model is in the passive state. The right hand side of the first order differential equation is converted to C++, substituting the signal buffer name for the signal name, and multiplying this buffer by the quantum size. The following is the Fourth-order Runge-Kutta method code for the low-pass filter presented above:

```
p1 = (1.0/(C*R))*(_vin*QuantumSize –
      (_vout*QuantumSize));
p2 = (1.0/(C*R))*(_vin*QuantumSize –
      (_vout*QuantumSize +
      sign(p1)*(HalfQuantumSize/2.0)));
p3 = (1.0/(C*R))*(_vin*QuantumSize –
      (_vout*QuantumSize +
      sign(p2)*(HalfQuantumSize/2.0)));
p4 = (1.0/(C*R))*(_vin*QuantumSize –
      (_vout*QuantumSize +
      sign(p2)*(HalfQuantumSize)));

h1 = HalfQuantumSize / p1;
h2 = HalfQuantumSize / p2;
h3 = HalfQuantumSize / p3;
h4 = HalfQuantumSize / p4;

h = 1.0/(1.0/(6.0*h1) + 1.0/(3.0*h2) +
    1.0/(3.0*h3) + 1.0/(6.0*h4));
```

The model then transitions to the active state for a time determined by *h*, which is calculated as in (4). The output function simply outputs the current state of the output buffer plus or minus one, plus one if the slope over the interval was positive, minus one if the slope over the interval was negative. The internal transition function similarly increments/decrements the state of the output buffer depending on the slope over the interval and then sends the model into the passive state.

During the coupled model generation, each of the integrator models converted during the model code and netlist generation (discussed in the following section), are instantiated and linked together. For each Integrator model instance, each port that represents a given endogenous quantity in the simultaneous statement is linked to all ports that represent that same quantity on itself and on all other Integrator model instances. All exogenous quantity and signal input ports are linked to their respective output ports on a process, component or signal model.

5. DEFINING CD++ COUPLED MODELS

Once the individual components are created, we need to convert designs written in sAMS VHDL into DEVS models that may be simulated in CD++. The application follows the dataflow illustrated in Figure 10.

The conversion begins with a check to ensure that the model is syntactically correct. Then, VHDL elaboration

yields to a description of the structure of each component in the design hierarchy. The architecture and entity description for each component in the design is parsed in order to produce a *netlist* (interconnected integrators, algebraic operators, processes, signals, etc.), which is used to generate CD++ model code for each of the processes. CD++ process models are then compiled into a model library.

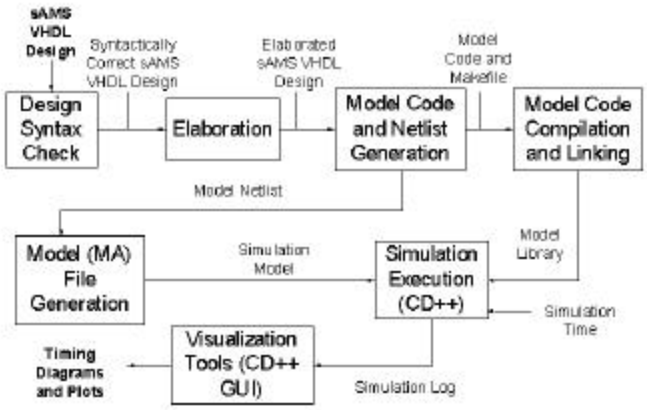

Figure 10. Simulation Dataflow

Following compilation, the netlist and model library are used by the model (MA) file generation process to yield a coupled CD++ model definition file. During this stage, sAMS VHDL models hierarchies are converted to CD++ coupled models. The components that constitute the design hierarchy must first be differentiated based on whether they are a basic or aggregate component. Basic components do not contain sub-component instances in their architectures, while aggregate components may have one or more. A dependency tree is generated: the leaves of this tree will be the basic components, while the branches will be aggregate components (the root will be the top-level model).

Figure 11 contains a CD++ coupled model definition for the sAMS VHDL design hierarchy of Figure 4, note the order of component declaration begins with the top level model and is followed by models that approach the leaves in the dependency tree. As we can see, there are two basic components: a digital clock (a coupled component built as the clock defined in Figure 6), and an integrator, built as in Figure 8.

sAMS VHDL sub-component instances are connected to the architecture in which they are instantiated as defined by the port map clause in their component instantiation statement. This clause will connect either a signal within the architecture, or a port on the architectures' entity definition to each of the ports on the component instance. In the case of a signal, the linking is termed *structural*, in the case of another port, the linking is termed *hierarchical*. In both cases the mode of the sub-component port specified in the *port* map clause must be determined prior to generating link statements in the coupled model definition. In structural links, if the ports mode is out, it is linked to the input port on the signal model specified in the clause; if the ports mode is in, the output port on the specified signal model is linked to it. In hierarchical links, if the sub-components port mode is out, it is linked to the component port; if the sub-components port mode is in, the component port is linked to it. Figure 11 illustrates all four of these cases.

```
entity LPF is
      port (terminal tout, tgnd: electrical);
end entity LFP;

   architecture top of LPF is
         signal clk : std_logic;
         signal vin : std_logic;
   quantity vout across tout to tgnd;

begin
   vout'dot = (1/(R*C))*(vin-vout);
   clk: entity clk
   port map (clk=>clk);
   vin<=clk;
end architecture top;
```

```
[top]
components : int@rkIntegModel clock
out : clk y
Link : y@int y
Link : y@int dydt@int
Link : out@clock clk
Link : out@clock vin@int

[int]
y0 : 0
dydt0 : 0
C : 1.0E-6
R : 1000

[clock]
components : inv@Process_Inv sig1@Signal
components : qm@QuantumMultiply
out : out
Link : out@sig1 in@inv
Link : out@inv  in@sig1
Link : out@sig1 in@qm
Link : out@qm   out

[sig1]
Transport_Delay : 00:00:1:000

[qm]
Transport_Delay : 00:00:00:000
Attenuation : 100
```

Figure 11. Hierarchical sAMS-VHDL Model

Once the complete model is defined and it has been translated, it can be simulated in CD++. The following figure shows the execution results for the filter using different input parameters.

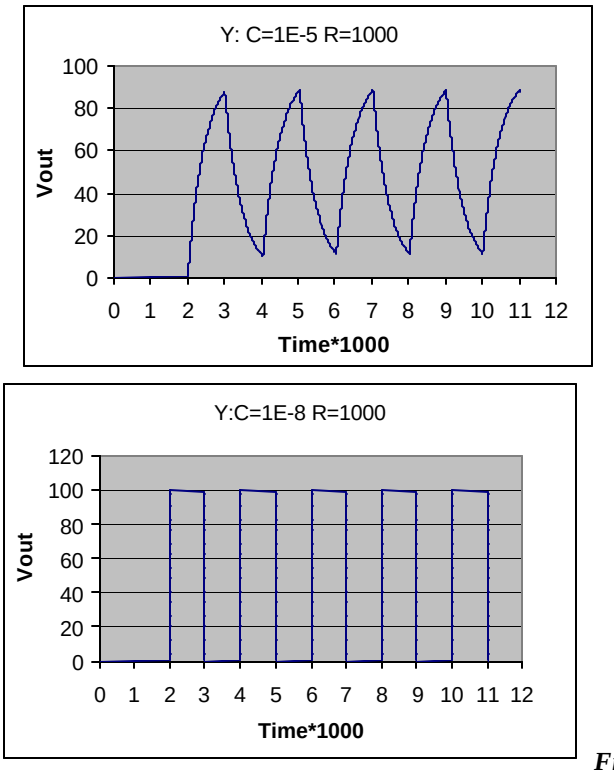

Figure 12. Simulation Results. Low-Pass Filter.

6. CONCLUSION

We showed the use of DEVS to facilitate simulation of mixed signal HDL models. In order to permit the execution of these models within a DEVS simulator, generic DEVS models and conversion procedures were required. Herarchical models written in sAMS-VHDL that utilize Processes, Signals and Simultaneous statements may be simulated in CD++ by elaborating the model, and converting the model hierarchy into an equivalent CD++ model.

The nature of DEVS permitted seamless integration of the model's components. Likewise, quantized DEVS permitted to integrate continuous signal models into a hierarchical model definition.

At present, we are extending sAMS-VHDL, and additional models and conversion procedures be developed. Type definition, generate blocks and signal attributes will ease model definition. The modularity of the CD++ models developed for this project will facilitate integration of new models.

ACKNOWLEDGEMENTS

This work has been partially supported by NSERC (National Science and Engineering Research Council of Canada), the Canadian Foundation for Innovation, and the IBM Eclipse Innovation Grants program.

REFERENCES

[1] KLOOS, C.; BREUER, P. Eds., "*Formal Semantics for VHDL"*. Dordrecht: Kluwer Academic Publishers, 1995.

[2] GIAMBIASI, N., ESCUDE, B., GHOSH, S. "*GDEVS: A Generalized Discrete Event Specification for Accurate Modeling of Dynamic Systems*," Transactions of the Society for Computer Simulation (SCS) International, Vol. 17, No. 3, September 2000, pp. 120-134, San Diego, CA.

[3] ZEIGLER, B.; KIM, T.; PRAEHOFER, H. "*Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*". Academic Press. 2000.

[4] CHRISTEN, E.; BAKALAR, K.; DEWEY, A.; MOSER, E. *"DAC'99 VHDL-AMS Tutorial"*. 36th Design Automation Conference, New Orleans, June 21-25, 1999.

[5] WAINER, G. "CD++: a toolkit to define discrete-event models". G. Wainer. *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[6] *IEEE 1076 Standard VHDL Language Reference Manual*, Design Automation Standards Committee (DASC). Piscataway: IEEE-SA Standards Board, 2000.

[7] ZEIGLER, B.P. "DEVS Theory of Quantization". DARPA Contract N6133997K-0007: ECE Dept., The University of Arizona, Tucson, AZ. 1998.

[8] PRESS, W.; FLANNERY, B.; TEUKOLSKY, S.; VETTERLING, W. "*Numerical Recipes in C: The Art of Scientific Computing"*. Cambridge University Press, 1992.

BIOGRAPHIES

Shaylesh Mehta received a B. Eng. in Computer Systems with High Distinction from Carleton University(2003). He is currently employed by MacDonald Dettwiler and Associates Space Missions Division, working on the Meteorological Station (MET) Instrument for the 2007 Phoenix Mars Lander Mission.

GABRIEL WAINER received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the Universidad de Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. He is Assistant Professor in the Dept. of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada). He was Assistant Professor at the Computer Sciences Dept. of the Universidad de Buenos Aires, and a visiting research scholar at the University of Arizona and LSIS, CNRS, France. He is author of a book on real-time systems and another on Discrete-Event simulation and more than 100 research articles. He is Associate Editor of the Transactions of the SCS, and the International Journal of Simulation and Process Modeling (Inderscience). He is Associate Director of the Ottawa Center of The McLeod Institute of Simulation Sciences. He has been awarded Carleton University's Research Achievement Award (2005-2006).