Transformation of VHDL Descriptions into DEVS Models for Fault Modeling and Simulation

Laurent Capocchi, Fabrice Bernardi, Dominique Federici, Paul-Antoine Bisgambiglia

To cite this version:

Laurent Capocchi, Fabrice Bernardi, Dominique Federici, Paul-Antoine Bisgambiglia. Transformation of VHDL Descriptions into DEVS Models for Fault Modeling and Simulation. IEEE Systems, Man and Cybernetics Conference (SMC03), Aug 2005, Washington, United States. pp.1205-1211. hal-00165462

HAL Id: hal-00165462
https://hal.science/hal-00165462

Submitted on 26 Jul 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Transformation of VHDL Descriptions into DEVS Models for Fault Modeling*

Laurent Capocchi, Fabrice Bernardi, Dominique Federici and Paul Bisgambiglia
University of Corsica, SPE Laboratory, UMR CNRS 6134
20250 Corte, France
{capocchi, bernardi, federici, bisgambi}@univ-corse.fr

Abstract – *We propose in this article an approach for the transformation of VHDL descriptions into DEVS models for an easy and fast fault simulation. VHDL allows description of the structure of a design, that is how it is decomposed into sub-designs, and how those designs are interconnected. The specification of the function of designs are performed using familiar programming language forms. One of the main problems is that today tools are unable to quickly and easily create and simulate fault models directly from the VHDL descriptions. A way to solve this problem is to encapsulate these descriptions in easily simulable and evolutive models. We propose to use the DEVS formalism to achieve this encapsulation.*

Keywords: VHDL descriptions, DEVS formalism, modeling, simulation, fault.

1 Introduction

Tests are a very important part in the digital systems design process. Fault modeling is one part of the tests that can be performed. We call "fault" the detection of an incorrect step, process or data definition in the design process [1, 2]. A fault model identifies targets for testing and makes analysis possible.

VHDL is an hardware description language widely used in the digital systems industry [3]. It allows description of the structure of a design, that is how it is decomposed into sub-designs, and how those designs are interconnected. The specification of the function of designs are performed using familiar programming language forms. One of the main problems is that today tools are unable to quickly and easily create and simulate fault models directly from the VHDL descriptions. A way to solve this problem is to encapsulate these descriptions in easily simulable and evolutive models. We propose to use the DEVS formalism to achieve this encapsulation.

Introduced by B.P. Zeigler in the early 70's, DEVS is a set-theoretic formalism that provides a mean of modeling discrete event systems in a hierarchical and modular way [4]. Using this formalism, we can perform more easy modeling in decomposing a large system into smaller component models with coupling specifications between them. DEVS defines two kinds of models: atomic models and coupled models. An atomic model is a basic model with specifications for the dynamics of the model. It describes the behavior of a component, which is indivisible, in a timed state transition level. Coupled models tell how to couple several component models together to form a new model. This kind of model can be employed as a component in a larger coupled model, thus giving rise to the construction of complex models in a hierarchical fashion.

Our choice for DEVS has been motivated by three main points. First, digital systems can be represented by discrete event systems, and DEVS is the leading formalism in discrete event modeling and simulation. Second, a great advantage of DEVS is that it separates the modeling and simulation parts in a very efficient way, since the simulator is built directly from the model. Last, DEVS uses a modular approach that allows an easy introduction of the fault models [5].

In a first section, we describe the VHDL language, and the DEVS formalism in a second one. The main part of this article is the third section. We propose a modeling approach for the VHDL descriptions based on the definition of four basic DEVS atomic models. Finally, we propose an example of modeling and simulation and we conclude this article.

2 VHDL Fault Modeling

VHDL allows to describe a digital structure following two types of descriptions: the structural description and the behavioural description. In the structural description, the design is described as a module with inputs and/or outputs. The electrical values on the outputs are some function of the values on the input. One way to represent the function of the module is to describe how it is composed of sub-models. However, in many cases, it is not appropriate to describe a module structurally. In the behavioural description, the function is described without reference to the actual internal structure of the

*0-7803-7952-7/03/$17.00 © 2003 IEEE.

module. We can find two forms for the behavioural description: the data flow form or the algorithmic form. In this paper, we study only this last form, since it has been demonstrated that any combinatory structure can be described using sequential statements [6].

A digital system in VHDL consists of a design entity containing other entities that are then considered components of the top-level entity. Each entity is modeled by an entity declaration and an architecture body. One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently.

```
entity NAME_OF_ENTITY is
  generic (generic_declarations);
  port (signal_names: mode type;
   signal_names: mode type;
   signal_names: mode type);
    :
end [NAME_OF_ENTITY];
```

The architecture body specifies how the circuit operates and how it is implemented. As discussed earlier, an entity or circuit can be specified in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above.

```
architecture arch_name of NAME_OF_ENTITY is
      -- components declarations
      -- signal declarations
      -- constant declarations
      -- function declarations
      -- procedure declarations
      -- type declarations
begin
      -- Statements
end architecture_name;
```

The statements in the body of the architecture make use of logic operators. Logic operators that are allowed are: and, or, nand, nor, xor, xnor and not. In addition, other types of operators including relational, shift, arithmetic are allowed as well.

The basis for sequential modeling is the process construct. A process statement is the main construct in behavioral modeling that allows you to use sequential statements to describe the behavior of a system over time. The syntax for a process statement is:

```
[process_label:] process [(sensitivity_list)]
[process_declarations]
begin
  list of sequential statements such as:
  signal assignments
  variable assignments
  case statement
  exit statement
  if statement
  loop statement
  next statement
  null statement
  procedure call
  wait statement
end process [process_label];
```

A process is declared within an architecture and is a concurrent statement. However, the statements inside a process are executed sequentially. Like other concurrent statements, a process reads and writes signals and values of the interface (input and output) ports to communicate with the rest of the architecture.

3 DEVS Formalism

DEVS formalism introduces two kind of models, atomic models from which larger ones are built, and coupled models (also called network of models) that connects those models in a hierarchical fashion [7]. Like in general systems theory, a DEVS model contains a set of states and transition functions that are triggered by the simulator.

A DEVS atomic model AM is a structure :

$$AM =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, t_a >$$

where:

- X: $\{(p, v)|$(p$\in$ input ports, v$\in$ Xp)$\}$ is the set of input ports and values for the reception of external events,
- Y: $\{(p, v)|$(p$\in$ output ports, v$\in$ Yp)$\}$ is the set of output ports and values for the emission of events,
- S is the set of internal sequential states,
- $\delta_{int} : S \rightarrow S$ is the internal transition function that will move the system to the next state after the time returned by the time advance function,
- $\mathbf{t}_a : S \rightarrow \mathbb{R}^+$ is the time advance function, that will give the life time of the current state (returns the time to the next internal transition),
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function that will schedule the states changes in reaction to an input event,
- $\lambda : Q \times X \rightarrow S$ is the output function that will generate external events just before the internal transition takes places.

The dynamic interpretation is the following:

- Q $= \{(s, e)|(s \in S, 0 < e < ta(s)\}$ is the total state set.

- e is the elapsed time since last transition, and s the partial set of states for the duration of $ta(s)$ if no external event occur.

- δ_{int} : the model being in a state s at ti, it will go into s', $s' = \delta_{int}(s)$, if no external events occurs before $ti + ta(s)$.

- δ_{ext} : when an external event occurs, the model being in the state s since the elapsed time e goes in s', $s' = \delta_{ext}(s,e,x)$.

  - The next state depends on the elapsed time in the present state.
  - At every state change, e is reset to 0.

- λ : the output function is executed before an internal transition, before emitting an output event the model remains in a transient state.

- A state with an infinite life time is a passive state (steady state), else, it is an active state (transient state). If the state s is passive, the model can evolve only with an input event occurrence.

The DEVS coupled model CM is a structure :

$$CM =< X, Y, D, \{M_d \in D\}, EIC, EOC, IC >$$

where:

- X is he set of input ports for the reception of external events,

- Y is the set of output ports for the emission of external events,

- D is the set of components (coupled or basic models),

- M_d is the DEVS model for each $_d \in D$,

- EIC is the set of input links, that connects the inputs of the coupled model to one or more of the inputs of the components that it contains,

- EOC is the set of output links, that connects the outputs of one or more of the contained components to the output of the coupled model,

- IC is the set of internal links, that connects the output ports of the components to the input ports of the components in the coupled models.

In a coupled model, an output port from a model $M_d \in D$ can be connected to the input of another $M_d \in D$ but cannot be connected directly to itself.

4 VHDL Descriptions Transformations

Our basic approach for the transformation is to associate each VHDL instruction with a DEVS atomic model, and each process with a DEVS coupled model. We defined four basic atomic models kinds. The first one is the "allocation model" kind that represents the VHDL allocation instructions. We can note that this model is able to handle time delayed allocations. The second is the "junction model" kind that represents how two atomic models are linked together. The third is the "conditional model" kind that represents the classical control instructions (if, case, for,...). Finally, the last one is the "parallel model" kind that manages the parallel execution of processes. All these basic models can be easily personalized and combined in order to create complex models.

4.1 The Allocation Model

The allocation model (Figure 1) is used to describe any type of allocation that does not take into account the notion of "delay".

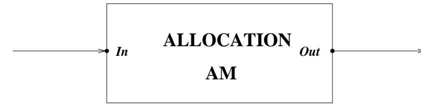

Figure 1: The Allocation Model

The associate atomic model is the following:

$$AM_{(alloc)} =< X, S, Y, \delta_{ext}, \delta_{int}, \lambda, ta >$$

where:

$$X = \{("In", v) | v \in V\}$$
$$Y = \{("Out", v) | v \in V\}$$
$$S = \{"active", "passive"\} \times \mathbb{R}_0^+ \times \{allocation\} \bigcup \emptyset$$

$$\delta_{int}(phase, \sigma, task) = ("passive", \infty, \emptyset)$$

$$\delta_{ext}(phase, \sigma, task, e, (In, message)) = ("active", 0, allocation),\ if\ phase = "passive"$$

$$\lambda(phase, \sigma, task) = (Out, message)$$

$$ta(phase, \sigma, task) = \sigma$$

Figure 2 presents the DEVS trajectories of an allocation model.

When an event occurs through the input port, the component becomes "active" using the δ_{ext} function. Then, the main task to achieve is to evaluate the assignment statement and to poke the output message using the λ function.

4.2 The Junction Model

The junction model (Figure 3) is used to describe sequential instructions like "end if, end elsif, end case".

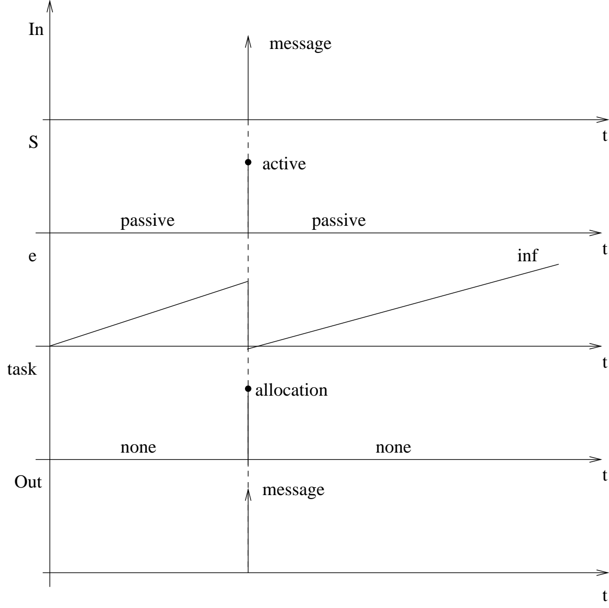

Figure 2: Allocation model trajectories

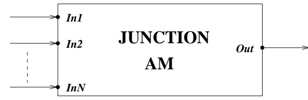

Figure 3: The Junction Model

The associate atomic model is the following:

$$AM_{(junction)} =< X, S, Y, \delta_{ext}, \delta_{int}, \lambda, ta >$$

where:

$X = \{(\text{"}In_0\text{"}, v), \cdots, (\text{"}In_N\text{"}, v) | v \in V\}$
$Y = \{(\text{"}Out\text{"}, v) | v \in V\}$
$S = \{\text{"}active\text{"}, \text{"}passive\text{"}\} \times \mathbb{R}_0^+ \times$
$\{\text{"}In_0\text{"}, \cdots, \text{"}In_N\text{"}\} \times V$

$\delta_{int}(phase, \sigma, input_port) = (\text{"}passive\text{"}, \infty, input_port)$

$\delta_{ext}(phase, \sigma, input_port, e, (p, v))$
if (phase = "passive") and (p $\in \{In_0, \cdots, In_N\}$)
$= (\text{"}active\text{"}, 0, p, v)$
else
$= (\text{"}passive\text{"}, \infty, input_port)$

$\lambda(phase, \sigma, input_port)$
if (phase = "active") *and* $(In_0, message)$
$= (Out, In_0)$
$\vdots$
if (phase = "active") *and* $(In_N, message)$
$= (Out, In_N)$

$ta(phase, \sigma, input_port) = \sigma$

If the component is in the "passive" phase and if an event occurs on an input port, the phase becomes "active" using the δ_{ext} function and the input port is selected. Then the λ function is actived and returns the selected input message.

4.3 The Conditional Model

The conditional model (Figure 4) occur in control structures like "if..then..else, when..else, with..select..when, case".

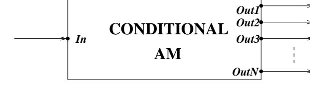

Figure 4: The Conditional Model

It is described by the following atomic model:

$$AM_{(conditional)} =< X, S, Y, \delta_{ext}, \delta_{int}, \lambda, ta >$$

where

$X = \{(\text{"}In\text{"}, v) | v \in V\}$
$Y = \{(\text{"}Out_0\text{"}, v), \cdots, (\text{"}Out_N\text{"}, v) | v \in V\}$
$S = \{\text{"}active\text{"}, \text{"}passive\text{"}\} \times \mathbb{R}_0^+ \times \{\text{"}Sw_0\text{"}, \cdots, \text{"}Sw_N\text{"}\}$

$\delta_{int}(phase, \sigma, Sw) = (\text{"}passive\text{"}, \infty, Sw)$

$\delta_{ext}(phase, \sigma, Sw, e, (In, message))$
if $(phase = \text{"}passive\text{"})$ *and* $(Sw = \text{"}Sw_0\text{"})$
$= (\text{"}active\text{"}, 0, \text{"}Sw_0\text{"})$
$\vdots$
if $(phase = \text{"}passive\text{"})$ *and* $(Sw = \text{"}Sw_N\text{"})$
$= (\text{"}active\text{"}, 0, \text{"}Sw_N\text{"})$
$= (\text{"}passive\text{"}, \infty, Sw)$ *other*

$\lambda(phase, \sigma, Sw)$
if $(phase = \text{"}active\text{"})$ *and* $(Sw = \text{"}Sw_0\text{"})$
$= (Out_0, message)$
$\vdots$
if $(phase = \text{"}active\text{"})$ *and* $(Sw = \text{"}Sw_0\text{"})$
$= (Out_N, message)$

$ta(phase, \sigma, Sw) = \sigma$

When an event occurs through the input port, the component becomes "active" using the δ_{ext} function. Then, the main task to achieve is to evaluate the conditional statement and to poke the output message into the selected port using the λ function. We can note that it exists as many output ports than values into the conditional statement.

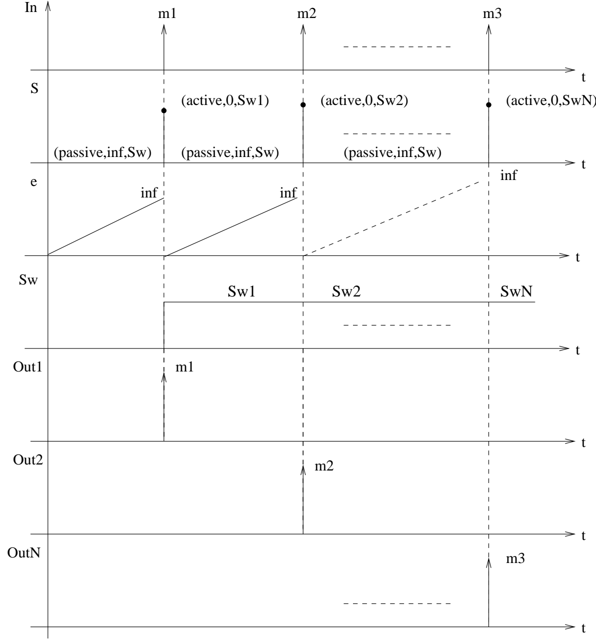

Figure 5: Conditional model trajectories

4.4 The Parallel Model

The parallel model (Figure 6) is used in order to synchronize the parallel execution of the processes. It is also used in order to manage the symbolic time.

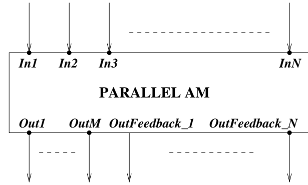

Figure 6: The Parallel Model

This model is described by the following atomic model:

$$AM_{(parallel)} = < X, S, Y, \delta_{ext}, \delta_{int}, \lambda, ta >$$

where

$X = \{(\text{"}In_0\text{"}, v), \cdots, (\text{"}In_N\text{"}, v) | v \in V\}$
$Y = \{(\text{"}OutFeedback_0\text{"}, v), \cdots, (\text{"}OutFeedback_N\text{"}, v) |$
$v \in V\} \cup \{(\text{"}Out_0\text{"}, v), \cdots, (\text{"}Out_M\text{"}, v) | v \in V\}$
$S = \{\text{"}active\text{"}, \text{"}passive\text{"}\} \times \mathbb{R}_0^+ \times \mathbb{R}_0^+$
$\times \{\text{"}physique\text{"}, \text{"}symbolique\text{"}\} \times \{\text{"}P0\text{"}, \text{"}P1\text{"}, ...\}$

$\delta_{int}(phase, \sigma, c, status, process)$
$if\ phase = \text{"}active\text{"}$
$= (\text{"}passive\text{"}, \infty, activeprocesses, \text{"}physical\text{"}, null)$
$else$
$= (phase, \infty, c, \text{"}physical\text{"}, null)$

$\delta_{ext}(phase, \sigma, c, status, process, e, (p, v))$
$if\ (c = 0)\ and\ (process! = [])$
$= (\text{"}active\text{"}, 0, length(process), status, process)$
$if\ (c = 0)\ and\ (process = [])$
$= (\text{"}passive\text{"}, 0, n_output_ports - 1, status, process)$
$if\ (c! = 0)$
$= (\text{"}passive\text{"}, \infty, c, status, process)$

$\lambda(phase, \sigma, c, status, process)$
$if\ phase = \text{"}active\text{"}$
$= \{(OutFeedback_0, msg), \cdots, (OutFeedback_N, msg)\}$
$else$
$= \{(Out_0, mesg), \cdots, (Out_M, mesg)\}$

$ta(phase, \sigma, c, status, process) = \sigma$

In this model, we add a new state variable called "c" (for "counter") that permits to count the number of active processes during a VHDL delta period. This variable is reduced each time a new message occurs during an active phase. The state variable "process" is a list defining the active processes. It is used in the λ function in order to activate the feedback output ports toward the processes. This list is built in the external transition function by testing the sensitive signal stationarity. Finally, the "status" variable allows to distinguish between the physical (with a VHDL physical simulation time) or symbolical (with a VHDL symbolical delta time) modes.

5 Experiments and Results

As an example, we present the modeling of a 8-bit register. This register is composed by three processes, each of them defining a "Wait" instruction. We chose to present this example since it shows the parallelism notion.

The VHDL description is the following code:

```
entity Register is
       port(DI: in BIT_VECTOR(1 to 8);
       STRB: in BIT; DS1:  in BIT;
       NDS2:  in BIT;
       DO: out BIT_VECTOR(1 to 8));
end Register;
Architecture behavior of Register is
signal reg :  bit_vector 1 to 8
signal enbld :  bit;

       strobe:  process(STRB)
       begin
             if (STRB = '1') then
             reg <= DI;
             endif;
       end process;
       enable:  process(DS1,NDS2)
       begin
             enlbd <= DS1 and not NDS2;
       end process;
       output:  process(reg,enbld)
       begin
```

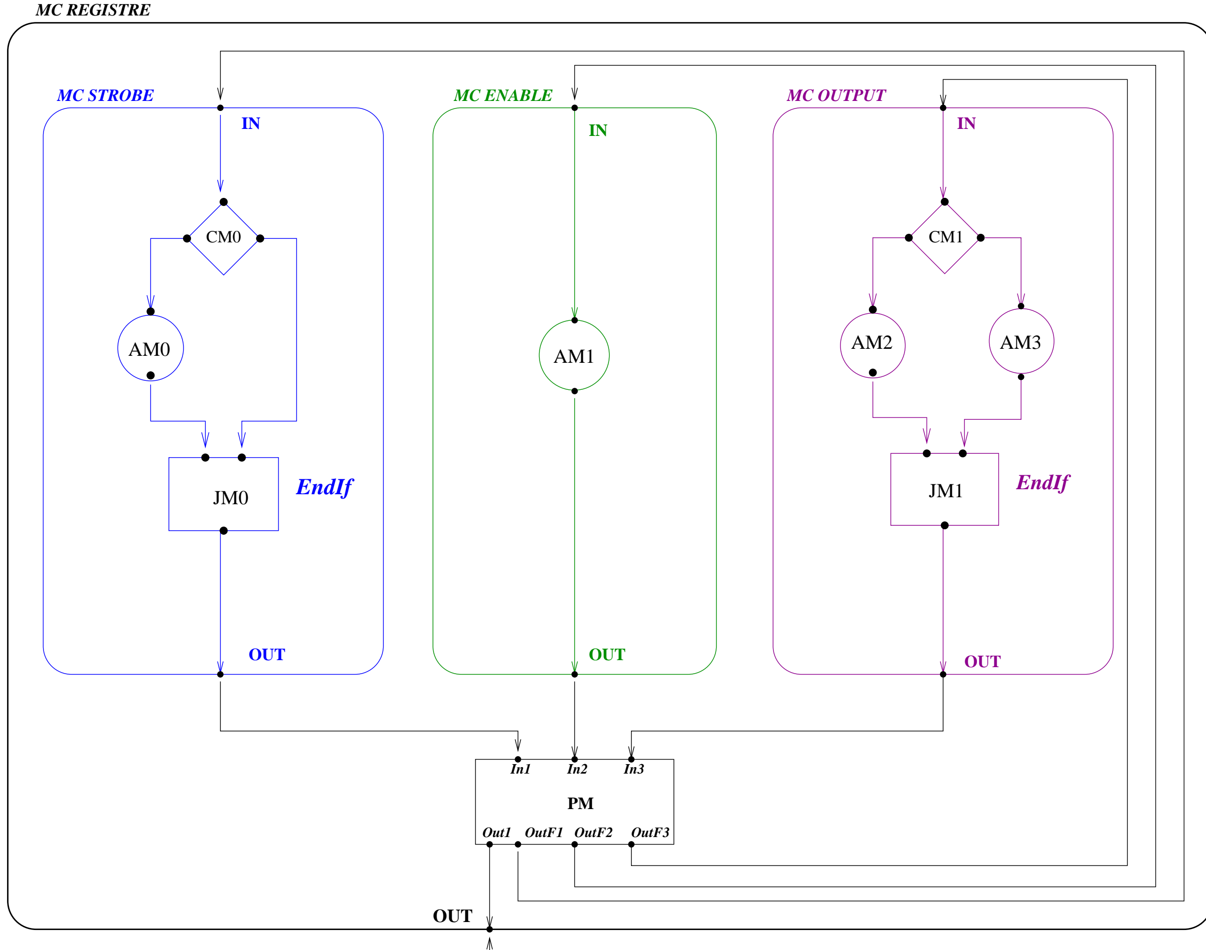

Figure 7: Modeling of a 8-bit Register

```
            if (enbld = '1') then
            DO <= reg;
            else DO <= 11111111
            endif;
      end process;
end behavior;
```

We can see in this code the three processes ("strobe", "enable" and "output").

The first process is modeled using three atomic models: a conditional model for the "if" statement (CM0), an allocation model for the assignment statement (AM0) and a junction model for the "endif" statement (JM0).

The second process is modeled using only one allocation model (AM1) since it defines only one instruction.

Finally, the third process is modeled using four atomic models: a conditional model (CM1) for the "if" statement, two allocation models (AM2 and AM3) and a junction model (JM1) for the "endif" statement.

At the highest level, the coupled model associated with the register can be described in the following way:

$$MC_{(register)} =< X, Y, D, M_d | d \in D, EIC, EOC, IC >$$

where

$X = \emptyset$
$Y = \{(OUT, v) | v \in V\}$
D $= \{MC_{strobe}, MC_{enable}, MC_{output}, MA\}$
EIC $= \emptyset$
EOC $= \{(MA, Out1), (MC_{register}, OUT)\}$
IC $=$
$\{((MA, OutF_1), (MC_{strobe}, IN)),$
$((MA, OutF_2), (MC_{enable}, IN)),$
$((MA, OutF_3), (MC_{output}, IN))$
$((MC_{strobe}, Out), (MA, In_1)),$
$((MC_{enable}, Out), (MA, In_2)),$
$((MC_{output}, Out), (MA, In_3))\}$

Figure 7 presents a representation of the modeled register. A circle corresponds to an allocation model, a diamond corresponds to a conditional model and a rectangle to a junction model. We can see a new atomic model called "PM" which is a parallel model. The main coupled model representing the register presents one output

CLK	D0	DI	STRB	enbld	reg	NDS2	DS1
0	0	00000000	0	1	1	0	0
0+	11111111	00000000	0	0	1	0	0
1	11111111	11100111	1	0	1	0	1
1+	11111111	11100111	1	1	11100111	0	1
2	11100111	00001111	1	1	11100111	1	1
2+	11100111	00001111	1	0	11100111	1	1
3	11111111	11110000	0	0	11100111	1	1
4	11111111	10101010	0	0	11100111	1	0
6	11111111	11111111	1	0	11100111	0	0
6+	11111111	11111111	1	0	11111111	0	0
8	11111111	00000000	0	0	11111111	1	0

Table 1: Simulation Results

port, but no input ports, since, for the needs of the simulation, we insert a new atomic model acting as an event generator.

The simulation has been performed using the PythonDEVS simulator package [8] developped at the McGill University and the results are showned in Table 1. This table presents the signal evolutions of the register, for each physical time step of the simulation. We show also the internal cycles of the simulation denoted with a "+".

We can also point out that the simulation time step is not regular. For example, we pass directly from time "4" to time "6", and from time "6" to time "8". This is because we can predict the simulation time for the process activations since we know the input events.

6 Conclusion and Perspectives

This paper was devoted to the presentation of our approach to transform VHDL descriptions into DEVS models. We defined four basic kinds of atomic models and associated a process to a coupled model. We only presented in this paper the modeling and the simulation of a 8-bit register, but we performed also a successful modeling and simulation of the ITC'99 benchmarks. We have one main perspective for this work: we want to add a fault model to our descriptions in order to be able to perform an easy and fast fault simulation [9]. We would also like to develop an integrated modeling and simulation environment dedicated to this task.

References

[1] V. Sieh, O. Tschche, and F. Balbach, "Comparing different fault models using verify," 1997.

[2] F. Celeiro, L. Dias, J. Ferreira, M. Santos, and J. Teixeira, "Vhdl fault simulation for defect-oriented test and diagnosis of digital ics."

[3] P. Ashenden, *The Designer Guide to VHDL*. Morgan Kaufmann Publishers, 2001.

[4] B. Zeigler, H. Praehofer, and T. Kim, *Theory of the Modeling and Simulation, 2nde Edition*. Academic Press, 2000.

[5] D. Federici, P. Bisgambiglia, and J. Santucci, "Behavioral fault simulation: Implementation and experiment results," in *Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications (DELTA '02)*, 2002. Christchurch, New Zealand.

[6] P. Ashenden, "The vhdl-cookbook," tech. rep., 1990.

[7] B. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models*. Academic Press, 1990.

[8] J. Bolduc and H. Vangheluwe, "A modeling and simulation package for classic hierarchical devs," tech. rep., 2002.

[9] E. Kofman, N. Giambiasi, and S. Junco, "Fdevs: A general devs-based formalism for fault modeling and simulation," in *Proceedings of the ESS 2000*, 2000. Hamburg, Germany.