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:

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.

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.

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
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.

```vhdl
entity NAME_OF_ENTITY is
  generic (generic_declarations);
  port (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.

```vhdl
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:

```vhdl
[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, Y, δint, δext, λ, ta >
```

where:

- **X**: \{(p, v)| (p \in \text{input ports}, v \in Xp)\} is the set of input ports and values for the reception of external events,
- **Y**: \{(p, v)| (p \in \text{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,
- **δint**: \(S \rightarrow S\) is the internal transition function that will move the system to the next state after the time returned by the time advance function,
- **ta**: \(S \rightarrow \mathbb{R}^+\) is the time advance function, that will give the life time of the current state (returns the time to the next internal transition),
- **δext**: \(Q \times X \rightarrow S\) is the external transition function that will schedule the states changes in reaction to an input event,
- **λ**: \(Q \times X \rightarrow S\) 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.
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”.

\[
\text{AM}_{\text{alloc}} = \langle X, Y, D, \{M_d \in D\}, EIC, EOC, IC >
\]

where:

- \( X \) is the 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.

4.2 The Junction Model

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

\[
\text{AM}_{\text{junction}} = \langle X, Y, D, \{M_d \in D\}, EIC, EOC, IC >
\]

where:

- \( X \) is the 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.
The associate atomic model is the following:

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

where:

- **X** = \{("In_0", v), \ldots, ("In_N", v) | v \in V\}
- **Y** = \{("Out_0", v), \ldots, ("Out_N", v) | v \in V\}
- **S** = \{"active", "passive"\} \times \mathbb{R}_+^+ \times \{"Sw_0", \ldots, "Sw_N"\}

\[ \delta_{int}(phase, \sigma, input_port) = ("passive", \infty, input_port) \]

\[ \delta_{ext}(phase, \sigma, input_port, e, (p, v)) \]

if (phase = "passive") and \( (p \in \{In_0, \ldots, In_N\}) \)
  = ("active", 0, p, v)
else
  = ("passive", \infty, input_port)

\[ \lambda(phase, \sigma, input_port) \]

if (phase = "active") and (In_0, message)
  = (Out, In_0)

\[ ta(phase, \sigma, input_port) = \sigma \]

When an event occurs through the input port, the component becomes "active" using the \( \delta_{ext} \) function and the input port is selected. Then the \( \lambda \) function is activated and returns the selected input message.

### 4.3 The Conditional Model

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

![Figure 4: The Conditional Model](image)

It is described by the following atomic model:

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

where

- **X** = \{("In", v) | v \in V\}
- **Y** = \{("Out_0", v), \ldots, ("Out_N", v) | v \in V\}
- **S** = \{"active", "passive"\} \times \mathbb{R}_+^+ \times \{"Sw_0", \ldots, "Sw_N"\}

\[ \delta_{int}(phase, \sigma, Sw) = ("passive", \infty, Sw) \]

\[ \delta_{ext}(phase, \sigma, Sw, e, (In, message)) \]

if (phase = "passive") and (Sw = "Sw_0")
  = ("active", 0, "Sw_0")

\[ \lambda(phase, \sigma, Sw) \]

if (phase = "passive") and (Sw = "Sw_N")
  = ("active", 0, "Sw_N")

\[ ta(phase, \sigma, Sw) = \sigma \]

When an event occurs through the input port, the component becomes "active" using the \( \delta_{ext} \) 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 \( \lambda \) function. We can note that it exists as many output ports than values into the conditional statement.
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.

This model is described by the following atomic model:

\[ AM_{\text{parallel}} = \langle X, Y, \delta_{\text{ext}}, \delta_{\text{int}}, \lambda, \tau > \]

where

- \( X = \{"In_0", v\}, \ldots, \{"In_N", v\}|v \in V\}
- \( Y = \{"OutFeedback_0", v\}, \ldots, \{"OutFeedback_N", v\}|v \in V\}
- \( S = \{"active", "passive"\} \times \mathbb{R}_0^+ \times \mathbb{R}_0^+ \times \{"physique", "symbolique"\} \times \{"P0", "P1", \ldots\}\)

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

- \( \delta_{\text{ext}}(\text{phase}, \sigma, c, \text{status, process, e, (p, v)}) \)
  - if \( (c = 0) \) and \( (\text{process} = []) \)

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 \( \lambda \) 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:

```vhdl
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;
    end if;
  end process;
  enable: process(DS1,NDS2)
  begin
    enbld <= DS1 and not NDS2;
  end process;
  output: process(reg,enbld)
  begin
```

Figure 5: Conditional model trajectories
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.
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 Python-DEVS simulator package [8] developed at the McGill University and the results are shown 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


