Modeling and Simulation of Logic Gates using DEVS

Maamar Hamri$^{1,2,3,4}$ and Nesrine Driouche$^{5}$

$^1$Aix Marseille Université, LSIS UMR 7296, 13397, Marseille, France
$^2$CNRS, LSIS UMR 7296, 13397, Marseille, France
$^3$ENSAI, LSIS UMR 7296, 13397, Marseille, France
$^4$Université de Toulon, LSIS UMR 7296, 13397, Marseille, France
$^5$Independent Software Researcher, Marseille, France

Keywords: Logic Gates, DEVS, Modeling and Simulation, Performances.

Abstract: Discrete event simulation becomes popular more and more and was applied successfully in many fields: medicine, robotics, etc. One of this field is digital circuits for which boolean logics is the basis of computation by designing logic gates. However such a paradigm does not consider the time basis. Consequently, the boolean logic paradigm can not design and simulate delays of circuits and stamped explicitly computations. In this paper, we propose to combine the boolean logic paradigm and Discrete EVent system Specification (DEVS) formalism for modeling and simulation logic gates. Using this approach, we are able to design complex network of logic gates by reusing and coupling basic ones and to analyze behavior through time.

1 INTRODUCTION

Digital circuits is viewed at conceptual level as networks of logic gates on which verification and test are possible. A basic logic gate has multiple inputs and only one output computed according to a truth table or a logic statement. Complex logic gates are built from basic ones and coupled using connectors. However, the boolean logic paradigm, alone, is not able to consider delay occurring in digital circuits due to used hardware materials (transistors) for the design.

Languages like VHDL, Verilog, etc. are well-known for designing logic gates, and checking formally and by simulation the behavior (Brown and Vranesic, 2003)(Palnitkar, 2003). However, we hope highlight the power of DEVS to design such gates.

Some works from DEVS focused on designing logic gates. (Tendeloo and Vangheluwe, 2013) proposed an approach to translate logisim© models in DEVS with keeping logisim semantics. This approach showed that DEVS simulator performs simulations of digital circuits more faster than logisim. (Campbell et al., ) proposed an approach to transform VHDL models to parallel-DEVS with an example based on an 8-bit register. However, the use of a synchronization model in order to schedule active processes slows down the simulation due to the additional exchanged messages inside the simulator. (Hamri et al., 2015) proposed an approach to design logic gates in GDEVS (Generalize Discrete EVent system Specification (Giambiasi et al., 2000)) in which input and output signals are linear segments instead of constants to represent accurately signals and state changes.

However, in this work, we propose a library of DEVS components built from the timed boolean logic paradigm to design logic gates, avoiding the use of an additional semantics from other paradigms like in (Tendeloo and Vangheluwe, 2013). So, we show that DEVS is able to design such gates using its own semantics and tools. On the other hand, designing input and output signals with constants (boolean 0 and 1) and states with constant piece-wise, even if such a design is less accurate than that proposed in (Hamri et al., 2015), allows comparisons with other tool like VHDL, Verilog, etc. using the same definitions on input and output signal, state changes, etc.

2 BACKGROUND

DEVS is one of the popular discrete event formalisms proposed in 70’s by Zeigler (Zeigler, 1976). The DEVS M&S framework separates clearly modeling
concerns from simulation ones. In fact, DEVS abstract simulator is useful to produce the behaviors of any model that respects the DEVS definitions. On the other hand, DEVS models are reused and coupled among them to make new DEVS models. Many research and practical works were realized around this formalism thanks to its powerful expressiveness. This formalism has many extensions: GDEVS (Giambiasi et al., 2000), parallel-DEVS (Chow, 1996), etc. and applications in different fields: forest fire spread (Hu et al., 2012), biological systems (Uhrmacher et al., 2005), etc.

2.1 DEVS Atomic Formalism

According to the literature on DEVS (Zeigler et al., 2000), the specification of a discrete event model is a structure, \( M \), given by:

\[
M = (X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, t_a)
\]

where

- \( X \) is the set of the external input events,
- \( S \) the set of the sequential states,
- \( Y \) the set of the output events,
- \( \delta_{\text{int}} \) is the internal transition function which defines the state changes caused by internal events, \( \delta_{\text{ext}} \) is the external transition function which specifies the state changes due to external events, \( \lambda \) is the output function, and the function \( t_a : S \rightarrow \mathbb{R}^+ \cup \infty \) represents the maximum length or the lifetime of a state. Thus, for a given state \( s \), \( t_a(s) \) represents the time during which the model will remain in state \( s \) if no external event occurs.

2.2 DEVS Coupled Formalism

DEVS promotes modular modeling to reduce the complexity of the system to describe. The DEVS coupled structure allows formalizing the modeled system in a set of inter-connected and reused components.

\[
MC = (X_{MC}, Y_{MC}, D_{MC}, M_{d \in D}, EIC, EOC, IC, Select)
\]

where

- \( X_{MC} \): set of external events.
- \( Y_{MC} \): set of output events.
- \( D_{MC} \): set of components names.
- \( M_d \): DEVS model named \( d \).
- \( EIC \): External Input Coupling relations.
- \( EOC \): External Output Coupling relations.
- \( IC \): Internal Coupling relations.
- \( Select \): defines a priority between simultaneous events intended for different components.

This formalism is proved by the closure under coupling property, which shows that a DEVS coupled model is a DEVS atomic one.

2.3 DEVS Simulator

The DEVS abstract simulator (see Figure 1) consists of a root-coordinator, which manages the simulation time, sub-coordinators which dispatch messages according to the specific couplings of the coupled model that attempt to simulate and basic simulators related to atomic models. Each process behaves according to the received messages from parent and child processes.

![DEVS abstract hierarchical simulator](image)

The classic structure of DEVS simulator is a hierarchical one, represented as a tree in which at top level is the root followed by the sub-coordinators created from DEVS coupled structure; then, at low level there are basic simulators related directly to the corresponding DEVS atomic models in order to execute the different functions \( \delta_{\text{int}}, \delta_{\text{ext}}, \lambda \) and \( t_a \). Figure 1 illustrates this structure and messages transiting from a process to another.

2.4 Simulators of Logic Gates

Designing digital circuits is based essentially on interconnected logic gates. These gates are a composition of the basic gates \textit{and}, \textit{or}, and \textit{not} in order to express the functional view of the circuits, and delay blocks to express the amount of times that a digital pulse crosses a physical device. A digital pulse is a series of waves reproduced periodically, i.e, a set of rise and fall events well stamped according to a global clock. In addition, such a pulse is limited to two voltage values (0 and 5 volts) which corresponds to 0 and 1 in logics.

Many software are dedicated to M&S logic gate like simulink®, VHDL®, Verilog®, logic gate simulator®, etc. that we can classify them into two main categories: educational and professional. Both of them use the M&S concepts noted above, modularity, reusing, connectors, etc. Educational software
2.5 Logic Gates in Verilog

Verilog standardized as IEEE1364, is a hardware description language used to model and simulate digital circuits (Wikipedia, 2015). It provides abstraction and reusing of primitives, blocks and modules to design new modules in hierarchical way. Many works showed the power of Verilog to design logic gates. In addition other works proposed methods to fix bugs and check timing conditions such as flip-flop set-up, hold violation, etc.

Verilog proposes different designs and implementation for logic gates. The modeler may design basic gates with or without ports. Time delay of logic gates is by default zero. However, the modeler may define a non-zero value for the pure delay (constructor delay) by using the operator 

\[ \# \text{delay} \]

In the following the Verilog code of the gates not and and:

//create a not gate
module logic_not (b, a);
// declare port signals
//b is output, a is input
output b;
input a;
not #1 n1(b,a);
endmodule

//create an and gate
module logic_and (c, a, b);
// declare port signals
output c;
input a,b;
//c is output, a, b are inputs
and #1 and1(c,a,b);
endmodule

From such basic modules, composite ones are built and for which statements may be added.

3 MODELING LOGIC GATES IN DEVS

Modeling logic gates in DEVS is easy thanks to its modeling concepts that ensures modularity, reuse, explicit time representation, etc. The modeler may enclose basic gates in DEVS atomic components then reuse them to design new components (nand, nor, flip-flop, etc.) by coupling imports and outputs of each reused components and the component under design according to DEVS coupled specification.

At low level, the function view of basic logic gates and, or and not, i.e how the output is computed from present inputs are logically described using the functions of DEVS atomic: \( \delta_{\text{ext}}() \), \( \delta_{\text{int}}() \), and \( \lambda() \). In fact \( \delta_{\text{ext}}() \) allows storing the inputs of the considered gate which schedules, then, an output according to the specified delay by calling the function \( \tau() \). Afterwards, the function \( \delta_{\text{int}}() \) allows the logic gate to change state from an active one to a passive one (waiting for a new input) enabling the sending of the computed output of the function \( \lambda() \).

Let us consider the logic gate and with two inputs and one output, and a pure delay \( d = 1 \) nanosecond (ns):

\begin{verbatim}
gate : and
parameter : d = 1 ns;
inports : in0, in1; outports : out;
variables states : bool a, b, y;
real sigma;
\end{verbatim}

\begin{verbatim}
\delta_{\text{ext}}(s, e, in0): a = in0;
y = min(a, b);
sigma = d;
\delta_{\text{ext}}(s, e, in1): b = in1;
y = min(a, b);
sigma = d;
\delta_{\text{int}}(): \sigma = \infty;
output(): send (out = y);
\tau(): return sigma;
endgate
\end{verbatim}

Note that, in our design for DEVS logic gates, we prefer the use of logic statements instead of truth tables to compute the output of each logic gate, in order to avoid a series of test to do that. In fact, we consider the following simple equations:

\begin{align*}
\text{not}(a) &= \neg a \\
\text{and} &= \min(a, b) \\
\text{or} &= \max(a, b)
\end{align*}

These elementary gates designed in DEVS are the basis of designing other well-known logic gate like nand, nor, xor, etc. The modeler reuses the necessary components and couples them through the imports and outputs to make the final component.

Let us consider the gate nand which consists of the gates and and not coupled in series. The following program describes the gate nand in DEVS:

\begin{verbatim}
gate: nand
inports : in0, in1;
outports: out;
\end{verbatim}
components: and, not;
couplings: couple(nand, in0, and, in0);
couple(nand, in1, and, in1);
couple(and, out, not, in1);
couple(not, out, hand, out);
endgate

Note that the modeler may define its own libraries to store designed components and may reuse them in hierarchical way furthermore. Moreover, some components may be designed differently. For example, the gate and with multiple imports may be designed as a composite gate based on a set of n gates and in series (see Figure 2).

![Figure 2: The composite gate and](image)

On the other hand, an elementary gate may be designed for such a gate and with n imports by using an algebraic function shown by the equation 4.

\[ \text{and}^n = \min(a_0, a_1, \ldots, a_n) \quad (4) \]

By analogy, the gate or with multiple imports may be designed in two different ways: by coupling a set of gates or in series; or by designing a new gate or\(^n\) where n is the number of inputs; and the computation of the corresponding output is based on the following equation 5:

\[ \text{or}^n = \max(a_0, a_1, \ldots, a_n) \quad (5) \]

These two designs are not really disconnected. In fact these two designs allow getting equivalent models. Knowing that the closure under coupling property of DEVS shows that each coupled model has an equivalent in DEVS atomic. Consequently, our two elementary gates and\(^n\) and or\(^n\) are obviously the result of the put in flat of the composite gates and and or with multiple imports.

Let us consider the gate composite and\(^n\) with multiple imports. According to the closure under coupling property, we are able to design a basic gate in DEVS atomic as follows:

gate : and
parameter : d = \times \text{delay};
// imports of the gate composite
inputs : in0, in1, \ldots, in_n;
// output of the gate composite
outputs : out;
variables states :
bool \times(a_0, b_i) = 1..n
real sigma, i=1..n;
\delta_{\text{ext}}(s, e, in_i):
\delta_{\text{ext}}(s, e, in_i); i = 1..n
\delta_{\text{int}}(a_0, e, in_i); i = 0
\delta_{\text{in}}(): \delta_{\text{out}}();
\delta_{\text{out}}(s, 0, \text{output}) = 1..n-1
output();
ta():
sigma = \min(\text{sigma}_1, \text{sigma}_2, \ldots, \text{sigma}_n)
return sigma;
endgate

By rewriting the functions \(\delta_{\text{ext}}\) and \(\delta_{\text{int}}\), we obtain:

\[ \delta_{\text{ext}}(s, e, \text{in}_i); b_i = \text{in}_i, i = 1..n \]

\[ \text{sigma} = \text{delay} \]

\[ \text{if } i = 0 \text{ then } a_0 = \text{in}_0 \text{ sigma} = \text{delay} \]

\[ \delta_{\text{out}}(): \text{sigma} = \begin{cases} \text{min}(a_i, b_i) \quad \text{if } i = 1..n-1 \\ \text{min}(a_i, b_i, b_i+1) \quad \text{if } i = n \end{cases} \]

The output of the gate \(\text{and}_{i+1}\) is:

\[ \lambda_{\text{out}}(a_{i+1}, b_{i+1}) \rightarrow \min(\text{sigma}_1, \text{sigma}_2) \rightarrow \min(\text{sigma}_1, \text{sigma}_2, \text{sigma}_3) \rightarrow \min(\text{sigma}, \text{sigma}_1, \text{sigma}_2, \text{sigma}_3) \rightarrow \min(\text{sigma}, \text{sigma}_1, \text{sigma}_2, \text{sigma}_3, \text{sigma}_4) \rightarrow \ldots \]

by recurrence, we have for the gate \(\text{and}^n\):

\[ \lambda_n : \text{min}(a_1, b_1, \ldots, b_{n-1}, b_n) \]

remains to rename the variables \(a_i\) to \(a_0\) and each \(b_i\) to \(b_i\), finally we obtain:

\[ \lambda_n : \text{min}(a_0, a_1, \ldots, a_n) \]

which corresponds to the output of the gate basic \(\text{and}^n\).

By using the same proof we can deduce also that the gate composite \(\text{or}^n\) may be designed with two different and equivalent designs.

4 DESIGNING LOGIC GATES IN DEVS SIMULATOR

Our logic gate components support any DEVS simulator (adevs, devssqljava, etc.) but we use our own DEVS simulator developed in Java to design these components, experiment it and show that it produce the right outputs at right times. The design of any composite logic gate follows the class diagram shown on Figure 3. The modeler should extend for each new logical gate the class DEVSCoupled, define imports and
outports, declare reused logical gates, and then define the coupling over them. Once the final logic gate is ready, the modeler may couple its inports with pulse generator component to emulate an input signal or init a scheduler of input events, then he can start the simulation.

![Figure 3: Object design of logic gates in DEVS.](image)

Now let us consider the well-known logic gate rs-latch defined by the following equation:

\[ Q' = Q \bar{R} + \bar{Q}S \]  

where \( S, R \) are the inputs, \( Q \) is the present output and \( Q' \) is the next output.

In DEVS, such a gate may be designed by four elementary gates: two or and two not; or in hierarchical way through two composite gates nor. In order to favour modularity, reusability and hierarchical design, we model the rs-latch by using two gates nor coupled correctly, even if the design based on elementary gates remains valid. The two possible designs are shown on Figure 4.

![Figure 4: Modeling a rs-latch in DEVS.](image)

A piece of code in our DEVS simulator implementing the gate rs-latch according to a hierarchical design, is shown below:

```java
public class SR extends DEVSCoupled {
    Nor nor1, nor2;
    Port in0, in1, out0, out1;
    public SR(){
        super();
        nor1 = new Nor("nor1");
        nor2 = new Nor("nor2");
        in0 = new Port("in0");
        in1 = new Port("in1");
        out0 = new Port("out0");
        out1 = new Port("out1");
        this.addInPort(in0);
        this.addInPort(in1);
        this.addOutPort(out0);
        this.addOutPort(out1);
        this.SubModels.add(this.nor1);
        this.SubModels.add(this.nor2);
        //sr (in0) --> nor1 (in0)
        this.addEIC(this.getInPorts().get(0),
                  this.nor1.getInPorts().get(0));
        ...
    }
}
```

However, to start simulation of the gate rs-latch, the modeler should define its init state. If this state is clearly known, the outputs of this gate belong to \( \{0,1\} \), but in some cases such a state may not be known because the outputs, that are also inputs, are not yet computed (cf. equation 6 and Figure 4). Consequently, a some interval time is necessary to compute the outputs and then set state variables of the init state. For that, we add a new behavior for each elementary gate to take into account the unknown state \( x \) according to the truth tables below:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>or</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
</tbody>
</table>

Now, let us consider the inputs \( s = 0 \) and \( r = 1 \) at time \( t = 0 \). The produced rs-latch outputs displayed on Figure 5, shows that our design using our simulator produces the right output events according to rs-latch boolean function at right time.

![Figure 5: Simulation of the rs-latch in DEVS.](image)
Moreover, the gate sr-latch may produce illegal output when \( s = r = 1 \). Our elementary gates respects DEVS encapsulation principle, so access to state variables are limited to elementary gate where those variables are declared. In order to avoid a not allowed state for the gate sr-latch, we may couple its output with a monitor component to forbidden undesirable outputs by checking the assertion \( \text{out}_0 \neq \text{out}_1 \) each time an output is sent out through the port \( \text{out}_0 \) or \( \text{out}_1 \). However, this solution even if it avoids undesirable outputs but it leads the sr-latch transitioning to an undesirable state. It is better to avoid undesirable inputs than handle undesirable outputs. By adding a monitor component handling the inputs \( s \) and \( r \), we guarantee that the sr-latch transition to a desirable state and send out desirable outputs. As conclusion, designing a valid sr-latch consists of using an additional component that checks whether the inputs are desirable and a gate sr-latch that design its logic function. This is an issue to handle desirable inputs for any logic gate.

5 PERFORMANCE COMPARISON

In order to show that DEVS is able to make simulations of logic gate as fast as verilog, we firstly give the simulation execution times of basic logic gates \( \text{and}, \text{or}, \text{not}, \text{nand} \) and \( \text{nor} \) in both adevs-2.8.1 and verilog-0.9.7. The input data simulation are identical for both simulations: same delay for all atomic gates and same input pulses; the simulation time is 4000 u.t (units of time). These simulations are carried out on personal computer DELL with CPU Intel®-Core2 Duo CPU E8400 - 3.00GHz × 2 where is set Ubuntu 14.0.4. The results are shown on Table 2.

<table>
<thead>
<tr>
<th></th>
<th>and</th>
<th>or</th>
<th>not</th>
<th>nand</th>
<th>nor</th>
</tr>
</thead>
<tbody>
<tr>
<td>adevs</td>
<td>16.0</td>
<td>15.6</td>
<td>9.2</td>
<td>24.8</td>
<td>17.2</td>
</tr>
<tr>
<td>verilog</td>
<td>23.6</td>
<td>22.0</td>
<td>18.0</td>
<td>24.0</td>
<td>23.2</td>
</tr>
</tbody>
</table>

By looking to results shown on Table 2, globally, the M&S of logic gates using adevs provides a concrete gain on execution time. So, DEVS simulation of logic gates is able to provide quick simulations as such as those provided by well-known tools in this field like verilog. In fact, the gate not designed in DEVS is distinguished by a small execution time comparing to other gates. The gate not has only one import, so it avoids the function \( \delta_{e_{\text{ext}}} \) to use the conditional statement if then to determine on which port the gate has received an event. This fact explains what the gates and or have considerable execution time for the same simulation data. These gates hold two imports and they should check on which port the event was received in order to update the right state variable and compute correctly the output. The composite gates nand and nor have the higher execution times due to fact that they involves each one two internal gates \( \{\text{and}, \text{not}\} \) and \( \{\text{or}, \text{not}\} \) respectively. It is quite that these composite gates present such execution times.

Now let us consider both design in adevs and verilog, the execution time of simulating basic gates and, or and not with adevs are less than those in verilog. However, the execution time of gates nand and nor in adevs and verilog are close. This lost on speed of adevs simulation is due to the architecture of DEVS simulation which introduces additional processes (coordinator) to simulate composite logic gates. Someone may believe that adevs loses on speed each time, for the simulation of more complex composite gates. For that we can use new architectures enhancing simulation process like those proposed by (Kim et al., 2000) and (Franceschini and Bisgambiglia, 2014) to reduce intermediary processes, and so the messages will cross short paths. What this suggests that adevs and DEVS simulator in C++ may provide interesting simulation execution times of logic gate close to Verilog.

Therefore DEVS as a discrete event formalism and its simulation software could compare with other software well-known and specialized in the field of logic gate application.

6 CONCLUSION

In this paper, we proposed a modular and hierarchical approach to design logic gates in DEVS. We used a simple language based on DEVS to design both atomic and composite gates. In this approach, we introduced logic statements well-known in logics to compute outputs of logic gates and the time basis to delay outputs with regard to inputs. In addition, we were led to re-design the behavior of basic gates in order to take into account the unknown state of logic latches that occurs at initialization of simulation. Note that our approach allows filtering inputs to avoid the non-allowed state for these latches by composition, i.e., by a simple DEVS component that sends out to latches only acceptable inputs. Recall that, the adevs and verilog simulations carried out for different logic gates provide execution times close for each other.

Moreover, the approach still opens to take into
account other types of delay: rise-fall, min-max, etc. which involve a simulation semantics different. DEVS and its extensions are able to deal with such delays; this point will constitute one of our future works. In addition, we will tackle the synthesis of composite logic gates (for example, three gates not in series is simplified to one gate not) and put in flat the network of gates automatically.

REFERENCES


