A Finite State Machine Modeling Language and the Associated Tools allowing Fast Prototyping for FPGA Devices

Bertrand Vandeportaele

A Finite State Machine Modeling Language and the Associated Tools allowing Fast Prototyping for FPGA Devices

Bertrand Vandeportaele
LAAS-CNRS, Université de Toulouse,
CNRS, UPS, Toulouse, France.
Email: http://homepages.laas.fr/bvandepo/

***Abstract*—The VHDL hardware description language is commonly used to describe Finite State Machine(FSM) models to be implemented on Field Programmable Gate Array(FPGA) devices. However, its versatility permits to describe behaviors that deviate from a true FSM leading to systems that are complex to prove, to document and to maintain.**

The purpose of this work is to propose a language and the associated tools to create FSMs through a dedicated and intuitive textual description. This language is inspired by the dot language used in Graphviz, a tool to define graphs, and adds all the necessary elements required to describe complex FSM models (using for instance memorized or non memorized actions and actions on states or transitions). Moreover some additional elements are proposed to enrich the standard FSM model such as the genericity that permits to define simultaneously multiple states, transitions or actions using a generative description.

A multi-platform open source JAVA program named FSMProcess [1] is introduced. Based on the ANTLR parser generator, it achieves the automatic generation of all the required .vhdl files (component, package, instantiation example and testbench) and a .dot file that is used to generate an always up-to-date graphical representation of the model (hence its documentation).

This tool also supports simple model checking and integration of additional VHDL code. It can be used conjointly with version control systems and is coupled with the open source GHDL simulator to allow fast prototyping. It can be used either with its Graphical User Interface either as a command line compiler for integration in makefiles.

I. Introduction

The Finite State Machine (FSM) model is a model of choice to describe a sequential system who do not require parallelism in its evolution (even if it allows parallelism for its actions or parallelism of execution of multiple models). This kind of model can be implemented using a behavioral description in VHDL language to configure logic devices (CPLD or FPGA) and achieve higher clock speed and lower resources requirements compared with the implementation on generic processors (CPU). While [2] proposes some systematic translation rules to obtain the VHDL code from the model, many other rules are observed in the literature and in source files from different projects. As the VHDL language is not specifically designed to implement FSM models, it offers degrees of freedom possibly leading developers to generate behaviors that deviate from the model.

In this paper, we propose to define and use a grammar that matches the FSM model structure, allowing a compact and minimalistic description of the model. This provide a kind of bijection between the model and its implementation, that allows to easily modify the model and thus permits to implement the model at the same time it is being constructed.

This work is inspired from SHDL (Simple Hardware Description Language) [3], a structural description language that aimed different goals but used the same philosophy: to describe a model in an adapted language and to export it to the targeted language using a compiler.

A compiler named FSMProcess [1], based on the ANTLR v4 [4] generic parser generator is introduced and distributed as open source. It features all the necessary functionality to design complex FSM models in an interactive way. This tools allows automatic inference of the interface of the model and generation of various VHDL files.

The paper is organized as follow: A simple example application is firstly introduced to demonstrate the usage of the FSM language to implement and test the model. Then, the processing pipeline involving other open source software is presented. Next, the FSM language is described and some simple model checking that are achieved on FSM models are presented. Finally, future improvements are proposed.

II. A simple application example

A simple system is used as an example to present both the basics of the language and the operation of the compiler. The chosen system's role is to count the angular position provided by two inputs (named A and B) connected to the outputs of an optical encoder based on two optical forks. This well known problem can be solved using the state machine model presented in the figure 1. This model pilots a counter which is either incremented or decremented at particular time when the INC and DEC signals are generated. The INC (resp. DEC) signal is generated once when the system is in the state 0 (resp. 1) and the A input exhibits a rising (resp. falling) edge.

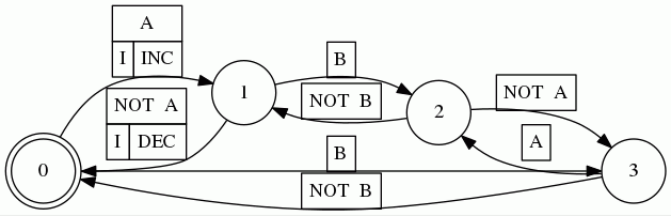

Fig. 1. The graphic representation of the example model. It is automatically generated by the proposed tool conjointly with graphviz.

A. Model Description

This basic model is described in the FSM language in an easy to interpret form by the following code, describing states, transitions, conditions and actions:

```
0->1?A:INC;
1->0?NOT A:DEC;
1->2?B;
2->1?NOT B;
2->3?NOT A;
3->2?A;
3->0?NOT B;
0->3?B;
```

B. Embedded VHDL code

This model is augmented with some embedded VHDL code through **pragma** directives. Readers familiar with VHDL easily recognize COUNT as the 16 bits output of a standard binary counter that can be synchronously reset by a SRAZ_CPT input.

```
#pragma_vhdl_entity{ COUNT : buffer
    std_logic_vector(15 downto 0);
        SRAZ_CPT : in std_logic; }#pragma
#pragma_vhdl_promote_to_buffer{INC,DEC}#pragma
#pragma_vhdl_architecture_post_begin{
--counter:
Process (ck, arazb)
begin
  if arazb='0' then count <= (others=>'0');
  elsif ck'event and ck='1' then
    if sraz_cpt='1' then
      count <= (others=>'0');
    elsif inc='1' then
      count <=count+1;
    elsif dec='1' then
      count <=count-1;
    end if;
  end if;
end process;
}#pragma
```

The proposed compiler is able to process this two concise parts of code and to automatically generate the following VHDL code, inferring inputs, outputs etc. and allowing the automatic generation of the graph shown in the figure 1. The real output of the compiler contains additional comments that ease the comprehension of the generated code but that are too long to display in this paper. Additional outputs displaying the current states number in binary are also automatically generated.

```
library ieee;
use    ieee.std_logic_1164.all;
use    ieee.std_logic_unsigned.all;
use    ieee.std_logic_arith.all;
entity quadencoder is
port (
    CK         : in   std_logic;
    ARAZB      : in   std_logic;
    STATE_NUMBER : out std_logic_vector( 1
        downto 0);
    A          : in   std_logic;
    B          : in   std_logic;
    SRAZ_CPT   : in   std_logic ;
    COUNT      : buffer std_logic_vector ( 15
        downto 0 ) ;
    DEC        : buffer std_logic;
    INC        : buffer std_logic);
end quadencoder;

architecture ar of quadencoder is
type fsm_state is (state_0, state_1, state_2,
    state_3);
signal current_state, next_state : fsm_state;
signal value_one_internal: std_logic;
begin
Process (ck, arazb)
begin
  if arazb='0' then count <= (others=>'0');
  elsif ck'event and ck='1' then
    if sraz_cpt='1' then
      count <= (others=>'0');
    elsif inc='1' then
      count <=count+1;
    elsif dec='1' then
      count <=count-1;
    end if;
  end if;
end process;
value_one_internal <='1';
process (CK, ARAZB)
begin
   if (ARAZB='0') then current_state
       <=state_0;
   elsif CK'event and CK='1' then
       current_state<=next_state;
   end if;
end process;
process (current_state, A, B)
begin
   case current_state is
    when state_0 => if ( ( A ) = '1' ) then
        next_state <= state_1;
                elsif ( ( B ) = '1' ) then
                    next_state <= state_3;
                else next_state <= state_0;
                end if;
    when state_1 => if ( ( NOT A ) = '1' )
        then next_state <= state_0;
                elsif ( ( B ) = '1' ) then
                    next_state <= state_2;
                else next_state <= state_1;
                end if;
    when state_2 => if ( ( NOT B ) = '1' )
```

```
        then next_state <= state_1;
                elsif ( ( NOT A ) = '1' )
                    then next_state <=
                    state_3;
                else next_state <= state_2;
                end if;
    when state_3 => if ( ( A ) = '1' ) then
        next_state <= state_2;
                elsif ( ( NOT B ) = '1' )
                    then next_state <=
                    state_0;
                else next_state <= state_3;
                end if;
   end case;
end process;
   DEC       <= '1' when ( (current_state =
       state_1) and ( ( ( NOT A ) ) = '1' ) )
       else
                '0';
   INC       <= '1' when ( (current_state =
       state_0) and ( ( ( A ) ) = '1' ) ) else
                '0';

  state_number <= "00" when ( current_state =
      state_0)
               else "01" when ( current_state
                   = state_1)
               else "10" when ( current_state
                   = state_2)
               else "11" when ( current_state
                   = state_3)
               else "11";
end ar;
```

The generated code exhibits the verbosity of the VHDL language. Compared with the model description using the FSM language, the VHDL code is much longer, harder to understand and modify, and prone to coding errors.

C. Simulation

The test of the model implementation is usually achieved through testbenches which imply to write non synthesizable VHDL code. The proposed compiler is able to generate automatically complete testbench files from a short description in the FSM file. For instance, the following concise FSM code allows the generation of a complete testbench.

```
#pragma_vhdl_testbench{
wait until (ck'event and ck='0' );
sraz_cpt<='1';
wait for ck_period;
sraz_cpt<='0';
A<='0';
B<='0';
wait for ck_period;
A<='1';
wait for ck_period*3;
A<='0';
wait for ck_period*3;
for i in 0 to 255 loop
A<='1';
wait for ck_period*3;
B<='1';
wait for ck_period*3;
A<='0';
wait for ck_period*3;
B<='0';
wait for ck_period*3;
END LOOP;
wait for ck_period*80;
}#pragma
```

The simulation consists in evaluating the testbench. The figure 2 shows the chronograms generated from it: while A and B inputs are sequentially activated, INC and DEC signal are generated by the state machine and the counter value evolves in the sequence 0,-1 (0xffff in 16bits signed representation),0,1.

D. Use of the generated component

A package file containing the generated component and another file containing an example of instantiation with default port map are automatically generated by the compiler.

III. Processing Pipeline

The proposed processing pipeline involves the open source FSM compiler [1] introduced in this paper and other free open source tools:

- Ghdl [5] : a fast VHDL simulator based on the compilation of the testbench to x86 code.
- GTKWave [6] : to display waveforms.
- Graphviz [7]: to generate bitmap and vector graphics of graphs with nodes and oriented arcs from a textual description in a .dot file.

A. Interactive development of single components

The FSM compiler is provided with a Graphical User Interface to develop the model interactively. The user can check in real time the model graphic representation and the various generated VHDL files. Some shell scripts (for linux and macos) and bat scripts (for windows) are provided to automatize the process of the files generation, allowing fast testbench execution through the use of Ghdl and GTKWave. The figure 3 recapitulates the different files being generated during the process, those in the second row being generated by the FSM compiler itself.

The image files generated from the .dot file supply an uptodate documentation about the model, and a text log file indicates errors and warnings about the compilation and pointing some possible model checking failures as described further.

This processing pipeline permits to ease the simulation of the model while the model is being described, ie. to allow the user to visualize the testbench execution interactively. To achieve that purpose, Ghdl is invoked and vcd file containing the waveform are generated. At the first time GTKWave is launched on this .vcd file, the user has to configure manually how the signals are organized. This settings are saved in a .sav file and are later reused even if additional signals are added or removed from the model, allowing a very effective mean of developing incrementally.

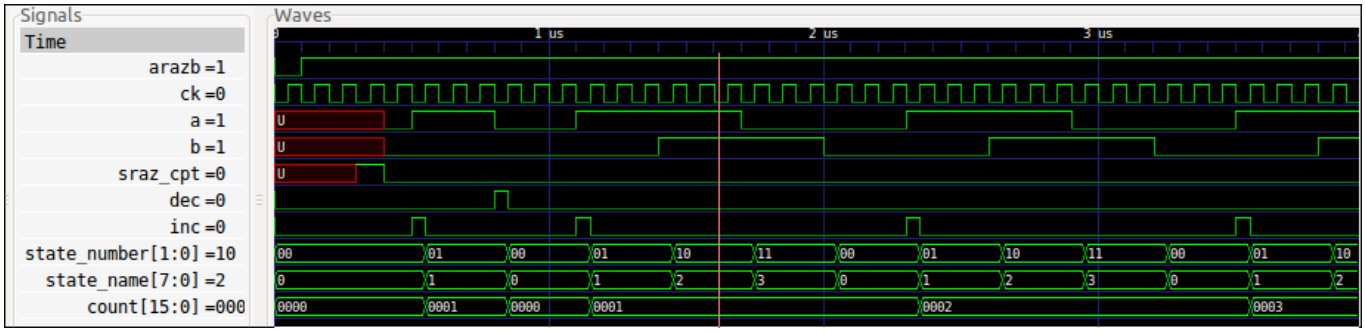

Fig. 2. Testbench simulation using GHDL. Screen capture from the GTKWave software.

Fig. 3. The Processing Pipeline.

B. Use in large projects

Large projects generally implies a large amount of files with dependencies, ie. some have to be processed firstly before others can be. This is commonly done through Makefiles that describe recipes to generate the different files up to the top level entity. The FSM compiler can be used in such a context, without the GUI, to generate silently its output to different files. Moreover, as the fsm files are essentially text files, they can reliably be managed by versioning tools like Git.

IV. The FSM language

A. Basic syntax

The very basic syntax of the proposed language is derived below. A more complete definition is given in the ANTLR v4 [4] grammar definition file (.g4) provided with the sources of the compiler but it requires the reader to master the ANTLR v4 grammar definition rules and this is out of the scope of this paper.

A Fsm model is described as a list of **instructions** whose order is generally not significant. These instructions are ended by `;`. Spaces and tabulations can be inserted to organize the text for the user's convenience and the language is case insensitive like the VHDL.

Some **comments** can be added anywhere in the code using the C syntax, ie. `//` operator for end of line comment and `/* */` operators for multiline comments.

A **state** is described simply by its name. Optionally, multiple actions can be attached to it, separated by the `:` operator.

```
origin_state1;
origin_state2:action_1:...:action_n;
```

A **transition between two states** is described using the `->` operator. An optional condition can be added using the `?` operator. A condition is a boolean expression using operators from the VHDL language. Transitions involving states that were not previously described will automatically generate the corresponding states. Optionally, multiple actions can be attached to a transition, separated by the `:` operator.

```
origin_state1->destination_state1;
origin_state2->destination_state2?condition;
origin_state3->destination_state3:action_1...
                              :action_n;
origin_state4->destination_state4?condition:
                    action_1:...:action_n;
```

When multiple transitions are defined from a given state, the conditions has to be mutually exclusive. To free the user from having to explicitly write mutually exclusive conditions, a **priority** operator `*` is used to set the priority level of each transition, the lower value being the higher priority. The default priority value is 1000.

```
origin_state->destination_state*priority?
                          condition:actions;
```

The actions supported by the model are of two kinds. First, the default actions involve **non memorized outputs** which are implicitly set to the 0 when not set. Otherwise they can

be set to 1 or to the value of a condition involving VHDL operators. These actions are defined using the optional **I** and the **,** operator as follow, multiple actions being separated by the **:** operator.

```
action_1:I,action_2:I,action_3=condition
```

The second kind of actions involves **memorized outputs** and more complex management. The initial value of such outputs at start up is defined in the same instruction than the asynchronous reset as shown further. Three kind of actions can be defined on memorized outputs: it can be reset to 0 using the **R** operator, be set to 1 using the **S** operator or memorize the value of a VHDL expression using the **M** operator. The reset and set actions can be conditioned by a VHDL condition.

```
S,action_1:R,action_2:M,action_3=expression
S,action_1=condition1:R,action_2=condition2
```

B. Advanced syntax

More advanced features of the syntax are described below, enriching the traditional model of the FSM.

The first described state is chosen by default as the **initial state** for the model (shown as a double circle in the graph). It is the active state at start up and when the **asynchronous reset** is triggered. The signal, active level and initial state can be redefined using the **=>** operator. **Initial values for memorized outputs** can also be defined (this can generate some synthesis problems for some FPGA depending on its reset circuitry).

```
=>state_1?asynchronous_reset_input_name,
    asynchronous_reset_active_value:
    output=initial_value...;
//example: =>state_1?CLRN,1: out1=IN3;
```

The **clock signal** can be changed from its default name CK to another one using the **/** operator.

```
/clock_input_name;
```

Actions can be defined once **whatever the current active state** of the model is to avoid the definition of the action in every states. These actions are defined using the **%** operator. This could for instance be used to describe a memorized output that is set during a particular transition and reset anytime under a condition. This also allows to generate easily a one clock cycle delayed version of a signal using a **M** type of action.

```
%action1;
```

Synchronous reset transitions can be defined using the **->** operator without state name on the left side. This allows to define in a single instruction potential transitions from any state including the destination state. Actions can be attached to such transitions and a VHDL condition is used to trigger the reset. A priority mechanism is used to determine which transition should be done, synchronous reset transitions having always an higher priority than standard transitions.

```
->destination_state1*priority?condition:actions;
```

C. VHDL code inclusion and interfacing with the FSM model

The inclusion of external language inside the FSM language is achieved through **pragma directives**. If such a directives uses arguments, its has to be ended by the following statement:

```
}#pragma
```

An input or an output generated from an action of the fsm model can be **promoted to a buffer** to be reused internally by the embedded VHDL code.

```
#pragma_vhdl_promote_to_buffer{output_name,...
```

The **automatic buffering** of the outputs being used as input in the fsm model can be authorized through:

```
#pragma_vhdl_allow_automatic_buffering
```

An input or an output generated from an action in the FSM can be **demoted to signal** to map it to an internal signal instead of an input or output signal:

```
#pragma_vhdl_demote_to_signal{output_name,...
```

Various VHDL code can be included in the FSM files, such as the inclusion of libraries, the addition of inputs or outputs, the definition of internal signals and the internal architecture description are inserted using the following statements:

```
#pragma_vhdl_pre_entity{
#pragma_vhdl_entity{
#pragma_vhdl_architecture_pre_begin{
#pragma_vhdl_architecture_post_begin{
```

As shown in the example, VHDL testbench is defined using:

```
#pragma_vhdl_init_testbench{
```

D. Genericity in the FSM model

Defining state names with an optional base name and ending with a number, the FSM syntax also permits multiple definitions in single instruction. For instance, **multiple sates** are defined using **(**, **to** and **)** operators:

```
base_state_name(number_begin to number_end);
```

Same actions occurring in multiple sates are defined by:

```
base_state_name(number_begin to
    number_end):actions...;
```

Mutliple transitions in cascade (from one state to the next one) using the **same condition** for the transitions are using the **#** operator:

```
#base_state_name(number_begin to
    number_end)?condition;
```

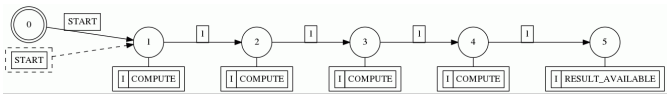

Fig. 4. Generated FSM model for generic parameter N=4.

Priorities and actions can be attached to such transitions using:

```
#base_state_name(number_begin to
    number_end)*priority?condition:actions....;
```

The concept of **generic parameters** used in the VHDL language is supported in FSM using:

```
#pragma_vhdl_generic_directive{
    parameter_name : parameter_type :=
    defaut_value; ...
```

The **effective value** of the generic parameter is set **at compile time** and this allows to adapt the generated component. For instances, N and M being two integer generic parameters, the width of an input B can be adapted using:

```
#pragma_vhdl_entity{ B : in
    std_logic_vector(N -1 downto M+2);}#pragma
```

This concept of genericity is pushed further in the FSM language allowing to define the model itself according to generic parameters. We just provide a short overview here due to lack of space. A FSM generic parameter is replaced by its actual value by the precompiler. Arithmetic operations achieved (for instance +,-,/,*, log...) on numeric generic parameters are evaluated if inserted in pragma:

```
#pragma_fsm_generic{ generic operation
    involving generic parameters....
```

This allows for instance to define transitions between states that depend on the parameter's value. Combined with the definition of multiple transitions in cascade, this permits to define FSM model whose portions of graphs correspond to different number of states. This is exploited in multiple examples provided with the compiler. For instance, this generic FSM model describes the sequencing of the successive division operations of two N bits operands and the figure 4 shows its graph for a value of N=4 :

```
=>0?RESETN;
->1?START;
0->1?START;
#(1 to #pragma_fsm_generic{N+1}#pragma);
(1 to #pragma_fsm_generic{N}#pragma):COMPUTE;
#pragma_fsm_generic{N+1}#pragma:RESULT_AVAILABLE;
```

V. Simple Model Checking

As the FSM model is processed by the compiler, some simple model checking is achieved to verify the coherence of the description. A brief overview is shown here. For instance, if multiple transitions are defined from a given state, the compiler can check if the corresponding conditions are mutually exclusive and generates a warning to ask the user to define priorities to disambiguate the model description.

As the FSM syntax is less verbose than VHDL, information have to be inferred from the model description. For instance used signals are detected as input, memorized or non memorized outputs, buffered signal as the model description is being compiled. The coherency between the different usages of the signals is being checked.

Inaccessible states are detected in the model. The results of the model checking are reported in a log file in the form of a textual description of errors and warnings.

VI. Conclusion and Future Works

A language to describe quickly and efficiently complex models of finite state machines has been introduced. Inclusion of some device dependent code (FPGA for logic devices) has also been proposed to facilitate the description of systems that contain logic alongside the FSM model. An open source compiler has been presented and a complete processing pipeline based on free open source software has been proposed to provide a complete tool allowing complex system design from the modeling and documentation to the functional simulation. This tool has a potential for industrial and educational purposes as it allows shorten development times, easier code maintenance and documentation and it is less prone to error thanks to a the use of a language that is specifically adapted to the model.

The FSM compiler [1] is provided with many examples showing more complex models that involve advanced functionality. For instance a simple soft core, multi clock cycles generic arithmetic operator (multiplication, division etc.), generic converters (for instance binary to BCD) and communication interfaces are provided.

In a near future, the model checking will be further investigated and the proposed compiler should integrate the export of the model to different languages (Verilog) and different hardware architectures (microprocessors using C language). This later (and simple to achieve) improvement would ease the co-design for architectures that integrate both microprocessor(s) and logic by describing the sequential parts of a whole system without requiring to firstly define on what kind of hardware they will later be executed.

An online version of the tool is also planed as it would allow potential users to try it without requiring specific installation on the client's computer.

References

[1] B. Vandeportaele, https://github.com/bvandepo/FSMProcess
[2] A. Nketsa and D. Delauzun, *Electronique numérique : systèmes numériques complexes*, Ellipses, 2012.
[3] J.C. Buisson, *Processeurs - Concevoir son micro-processeur - Structure des systèmes logiques*, Ellipses, 2006.
[4] T. Parr, *The Definitive ANTLR 4 Reference*, Pragmatic Bookshelf, 2013.
[5] T. Gingold, *GHDL*, http://ghdl.free.fr/.
[6] *GTKWave*, http://gtkwave.sourceforge.net/.
[7] *Graphviz*, http://www.graphviz.org/.