GenFSM, a Finite State Machine Generation Tool

Codrin PRUTEANU [1], Cristian-Győző HABA [2]
[1]*AsicAhead International S.R.L.,*
Calea Rahovei, 266-268, Corp 61,
Sector 5, Bucharest
codrin.pruteanu@gmail.com
[2]*Faculty of Electrical Engineering*
"Gh. Asachi" Technical University, Iasi
Bd.D. Mangeron, nr.53, Iasi
cghaba@ee.tuiasi.ro

***Abstract*—In this paper we present *GenFSM* - a tool that was designed by the authors to generate a number of completely or incompletely specified finite state machines. *GenFSM* will receive at the input a list of arguments regarding the required number of internal states, the number of inputs and outputs, and also the desired output format. *GenFSM* will generate a specified number of state machines that are described as state transition tables for academic use, or in Verilog format as a standard hardware description language in the CAD industry, in order to be used for portability and integration with another tools for design or testing.**

***Index Terms*—finite state machine, state transition table, logic synthesis, system-level design, Verilog**

I. INTRODUCTION

Similarly to the Finite Automata model used in computer design, a finite state machine (FSM) model finds its widest use in hardware engineering. As a hardware development design pattern, FSM occurs in many hardware solutions, but unlike most of its pattern counterparts, FSM requires substantial time for manual implementation. The really critical part of the manual implementation is usually in its increasing cost of change and maintenance. All of the above considerations eventually converge to a necessity of an automatic fault-free FSM generation tool.

In this paper, we describe our goal to develop a software tool, ***GenFSM***, which would be able to:

- Automatically produce completely or incompletely specified FSM implementation based on the configuration arguments on its input
- The output format of the generated FSMs can be described as state-transition tables (*Kiss* level)[9], or in *verilog* (behavioral level)
- Generate input data for any other development or testing tool, without posting any compatibility issues
- Completely transparent to the user, but still, fault-free and producing completely described FSMs.

GenFSM can be used for its primary purpose – to generate FSMs with arbitrary configuration, as well as, to help building powerful tools for hardware design, implementation, logic synthesis optimization, testing, verification, etc [3]. Some examples of such applications are: graphical automata modeling, model creation for model-based verification, model-based code synthesis, digital design, etc.

In the next section we discuss about FSMs theoretical backgrounds and related work. We formulate a solution and propose an example in chapter III. Also, we discuss about the results and take some conclusions in chapter IV.

II. PROBLEM FORMULATION

A FSM can be classified as deterministic or non-deterministic, and completely or incompletely specified. A completely specified state machine M (called CSFSM), can be described by a tuple $\boldsymbol{M = (I, S, O, \delta, \lambda)}$, where $\boldsymbol{I}$ is a set of primary inputs, $\boldsymbol{S}$ is a set o state symbols, $\boldsymbol{O}$ is a set of primary outputs, $\boldsymbol{\delta: I \times S \rightarrow S}$ is the next state function, and $\boldsymbol{\lambda: I \times S \rightarrow O}$ is the output function for Mealy machines or $\boldsymbol{\lambda: S \rightarrow O}$ is the output function for Moore machines. Moore machines can be considered a special case of Mealy machines. Therefore the set of theories and methods used for Mealy machines will be also applicable to Moore machines. In the case of incompletely specified FSMs (called ICSFSM), $\boldsymbol{I}$ represents a finite set of inputs, $\boldsymbol{S}$ is a finite non-empty set of internal states, $\boldsymbol{O}$ is a finite set of outputs, $\boldsymbol{\delta : I \times S \rightarrow 2^S}$ is the next-state function, and $\boldsymbol{\lambda : I \times S \rightarrow 2^O}$ is the output function.[5] A further distinction is between deterministic (DFSM) and non-deterministic (NDFSM) finite state machine. In deterministic FSM, for each state there is exactly one transition for each possible input. In non-deterministic automata, there can be none or more than one transition from a given state for a given possible input.[6] This distinction is relevant in practice, but not in theory, as there exists an algorithm which can transform any NDFSM into an equivalent DFSM, although this transformation typically significantly increases the complexity of the state machine. The FSM with only one state is called a combinatorial FSM and uses only input actions. This concept is useful in cases where a number of FSM are required to work together, and where it is convenient to consider a purely combinatorial part as a form of FSM to suit the design tools. A FSM can be described in a tabular form by a state transition table (STT), in a graphical form by a state transition graph (STG), or by using RTL code. ***GenFSM*** reads a number of arguments at the input and generates at the output FSMs described in *Kiss* or *verilog* code. The states of an FSM are listed as symbolic state names. Each symbolic state name has a corresponding binary value. Symbolic minimization performs the logic

minimization phase before the FSM state encoding.[4] Symbolic minimization was implemented by DeMicheli in *KISS* [2] in 1985. Some of the shortcomings of *KISS* are addressed in its successor NOVA [8]. NOVA takes more efficient and flexible approach to constraints satisfaction, representing it as a graph embedding problem and solving in several, heuristic strategies producing superior results and offering quality/runtime trade-offs [1].

III. PROBLEM SOLUTION

GenFSM is using a standard FSM description template, where the next state and the output functions for each generated FSM are depending on the primary inputs and the present states of that FSM. (See Figure 1) [7]

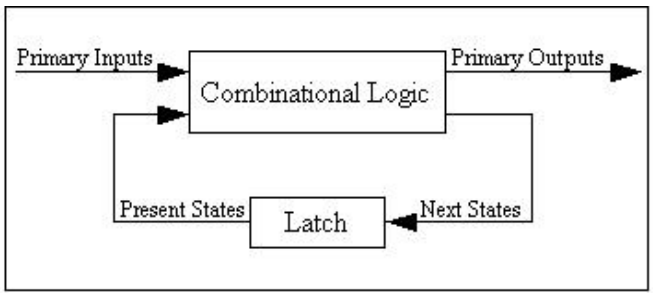

Figure 1. General sequential circuit.

We take an example where, by using ***GenFSM***, we generated 2 FSMs. For the completely-specified FSM, the STG description can be seen in Figure 2 and the STT description can be seen in table I. For the incompletely-specified FSM, the STG description can be seen in Figure 3 and the STT description can be seen in table II.

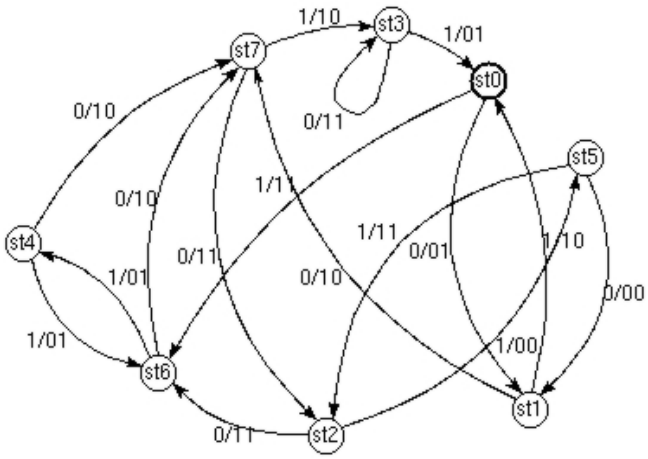

Figure 2. STG for a completely specified FSM.

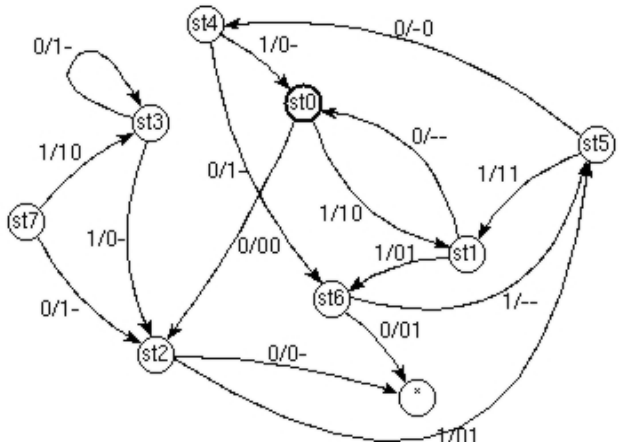

Figure 3. STG for a incompletely specified FSM.

As we can see in the graphical representation, in the case of ICSFSM, ***GenFSM*** is able to generate an incompletely specified value for the output function and also an undefined state for the next state function, which can be observed in the STG by a star.

TABLE I. STT FOR A COMPLETELY-SPECIFIED FSM

Input	Pres State	Next State	Output
0	St0	St1	0 1
1	St0	St6	1 1
0	St1	St7	1 0
1	St1	St0	1 0
0	St2	St6	1 1
1	St2	St5	0 0
0	St3	St3	1 1
1	St3	St0	0 1
0	St4	St7	1 0
1	St4	St6	0 1
0	St5	St1	0 0
1	St5	St2	1 1
0	St6	St7	1 0
1	St6	St4	0 1
0	St7	St2	1 1
1	St7	St3	1 0

TABLE II. STT FOR AN INCOMPLETELY-SPECIFIED FSM

Input	Pres State	Next State	Output
0	St0	St2	0 0
1	St0	St1	1 0
0	St1	St0	- -
1	St1	St6	0 1
0	St2	*	0 -
1	St2	St5	0 1
0	St3	St3	1 -
1	St3	St2	0 -
0	St4	St6	1 -
1	St4	St0	0 -
0	St5	St4	- 0
1	St5	St1	1 1
0	St6	*	0 1
1	St6	St5	- -
0	St7	St2	1 -
1	St7	St3	1 0

For portability, ***GenFSM*** was designed to produce FSMs by respecting the *Kiss* or *verilog* language syntax. The STT representation was considered as the basic input to obtain the *Kiss* language. (Figure 4)

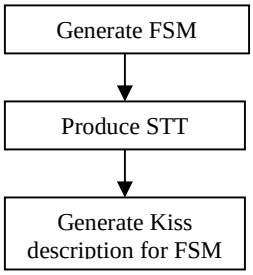

Figure 4. Design flow used to generate *Kiss* FSM description.

The STG representation was used to describe the generated FSMs by using *verilog* code. (Figure 5)

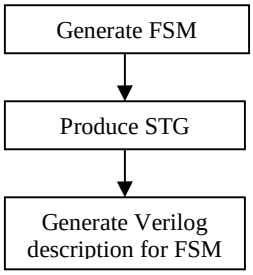

Figure 5. Design flow used to generate *verilog* FSM code.

The results obtained by ***GenFSM*** for the ICSFSM example using the kiss-level description can be seen in Fig. 6. *Kiss* is a tabular format, where each row has four entries: input field, present state field, next state field and output field. There are as many rows as transitions in the state graph of the FSM.

```
.i 1
.o 2
.s 8
.p 16
.r st0
0 st0     st2     00
0 st1     st0     --
1 st1     st6     01
0 st2     *       0-
…….
0 st6     *       01
1 st6     st5     --
1 st7     st3     10
.e
```

Figure 6. Kiss file written by ***GenFSM*** .

In the case that the output of the generated FSM is described by using verilog code, we might have 2 different approaches. ***GenFSM*** defines the internal states from the STG description as verilog parameters and is using a binary encoding of states. The main difference between the completely and the incompletely-specified FSM approaches is that in the case of incompletely-specified FSM, a new state is added to the states defined in the FSM description, which is called 'statex'. This state is used to describe an undefined next-state of the generated FSM. The main sequential loop is given by the following verilog sequence:

```
always @(posedge clock)
 if (reset)
  state = st0;
 else
  state = nextstate;
```

The next state and the output functions are defined in a *verilog* 'always' sensitivity list that depends on the values of the current state and the inputs. A small fragment of the verilog code that describes the next-state function and the output function is shown below:

```
always @(state or in)
 begin
  out = 2'b00;
  nextstate = st0;
  case (state)
   st0: casex (in)
         1'b0: begin
                    nextstate = st5;
                    out = 2'b1x;
                end
         1'b1: begin
                    nextstate = statex;
                    out = 2'bxx;
                end
        endcase
```

As we can check in the verilog description, the second sequential loop is accomplished by using the 'always' keyword which detects when the values of the present state or the current input are changing. If the present state has been changed, then we check what the next state is and what output value of the FSM has been reached. As can be seen into the Appendix A full example, the ***GenFSM*** handles the case of "unspecified value" by using "casex" statement in the case of incompletely-specified FSMs. Each node has a single-output logic function associated with it and each feedback loop contains at least one latch (flip-flop). Each signal has only a single driver, and either the signal or the gate which drives the signal can be named without ambiguity.

IV. RESULTS AND CONCLUSIONS

GenFSM proved its functionality as a development tool during the PhD. research of the 1st author and has been used for its primary purpose – to generate FSMs from arbitrary configuration that have been used as primary input data for other FSM optimization tools designed by the authors. CAD designers require a fast and direct way to convert the original design into hardware description languages, such as HDL code, in order to simulate and implement it. One of the main advantages of using such a free "personal" tool was that it provided access to academic and independent research without the need of a commercial tool, while giving the advantage of generating the FSMs into a portable, standard format (*kiss* or *verilog*), that can be well used by any other hardware tools for digital design and testing.

REFERENCES

[1] Brayton K. R., Sangiovani-Vincentelli A., SIS: A system for sequential circuit synthesis, Electronic Research Laboratory, Memorandum No. UCB/ERL M92/41, University of California, Berkeley, CA 94720, 4 May 1992.

[2] De Micheli G., Brayton R. K., and A. Sangiovanni-Vincentelli. Optimal state assignment for Finite State Machines. IEEE Trans. on CAD, pages 269-284, 1985.

[3] De Micheli G., Synthesis and Optimization of Digital Circuits, Stanford University, McGraw-Hill, 1994.

[4] Devadas S., Newton A. R., Decomposition and factorization of sequential finite state machines, IEEE Trans. on CAD, Vol. 8, No. 11, November 1989, pp. 1206-1217.
[5] Hartmanis J. and Stearns R.E., Algebraic Structure Theory of Sequential Machines, Prentice Hall, 1966
[6] Muntean I., Finite Automata Synthesis, Editura Tehnica, Bucharest, Romania, 1997.
[7] Pruteanu C., Galea D., Haba C.G., Global Optimization in Complex Circuits Design, Proceedings on 7th IEEE International Symposium on Signals, Circuits, Systems (ISSCS 2005), vol.2, ISBN 0-7803-9029-6, IEEE Catalog Number: 05EX1038, July 14-15, 2004, Iasi, Romania;
[8] Villa T. and Sangiovanni-Vincentelli A. NOVA: state assignment of Finite State Machines for optimal two-level logic implementation. IEEE Trans. on CAD, pages 905-924, 1990
[9] Villa T., Gitanjali S., Shiple T., VIS User's Manual, The VIS Group: University of California, Berkeley, University of Colorado, Boulder, Now at Lattice Semiconductor, 1996

APPENDIX A

We show an example of an incompletely-specified FSM that was described as a STT, in *Kiss* level format, where ***.i*** represents the number of inputs, ***.o*** the number of outputs, ***.s*** the number of internal states, ***.p*** the number of products and ***.r*** the reset (original) state.

```
# Kiss file written by GenFSM
.i 1
.o 2
.s 8
.p 16
.r st0
0 st0     st2     00
1 st0     st1     10
0 st1     st0     --
1 st1     st6     01
0 st2     *       0-
1 st2     st5     01
0 st3     st3     1-
1 st3     st2     0-
0 st4     st6     1-
1 st4     st0     0-
0 st5     st4     -0
1 st5     st1     11
0 st6     *       01
1 st6     st5     --
0 st7     st2     1-
1 st7     st3     10
.e
```

Here we have an example of the verilog code for another incompletely-specified FSM generated by *GenFSM.*

```
// Verilog file written by GenFSM
module ex1_1_8_2_1205165646 (reset, clock, in, out);
input reset, clock;
input in;
output [1:0] out;
reg [1:0] out;
reg [2:0] state, nextstate;
parameter st0 = 0,
          st1 = 1,
          st2 = 2,
          st3 = 3,
          st4 = 4,
          st5 = 5,
          st6 = 6,
          st7 = 7,
          statex = 3'bxxx;
always @(posedge clock)
 if (reset)
  state = st0;
 else
  state = nextstate;
always @(state or in)
 begin
  out = 2'b00;
  nextstate = st0;
  case (state)
   st0: casex (in)
         1'b0: begin
                  nextstate = st5; out = 2'b1x;
                 end
         1'b1: begin
                  nextstate = statex; out = 2'bxx;
                 end
        endcase
   st1: casex (in)
         1'b0: begin
                  nextstate = st0; out = 2'b00;
                 end
         1'b1: begin
                  nextstate = st1; out = 2'b01;
                 end
        endcase
   st2: casex (in)
         1'b0: begin
                  nextstate = st3; out = 2'b10;
                 end
         1'b1: begin
                  nextstate = st2; out = 2'b0x;
                 end
        endcase
   st3: casex (in)
         1'b0: begin
                  nextstate = st4; out = 2'b11;
                 end
         1'b1: begin
                  nextstate = st2; out = 2'b0x;
                 end
        endcase
   st4: casex (in)
         1'b0: begin
                  nextstate = statex; out = 2'bxx;
                 end
         1'b1: begin
                  nextstate = st0; out = 2'b00;
                 end
        endcase
   st5: casex (in)
         1'b0: begin
                  nextstate = st1; out = 2'b01;
                 end
         1'b1: begin
                  nextstate = st6; out = 2'bx0;
                 end
        endcase
   st6: casex (in)
         1'b0: begin
                  nextstate = statex; out = 2'bxx;
                 end
         1'b1: begin
                  nextstate = st6; out = 2'bx0;
                 end
        endcase
   st7: casex (in)
         1'b0: begin
                  nextstate = st1; out = 2'b01;
                 end
         1'b1: begin
                  nextstate = st4; out = 2'b11;
                 end
        endcase
  endcase
 end
endmodule
```