State Machine Coverage Using Specman

Jacob Joseph

1.0 Introduction

FSM (Finite State Machine) coverage is very important in coverage driven functional verification, because of state machines can have many branches and multiple functional paths and any hidden path (a functional path that missed in verification or a path that unintentionally introduced during implementation phase) in design can cause serious misbehave in functionality as well can create deadlock (system is not able to recover from a particular state by itself, even though the intended stimulus is there).

Coverage helps to avoid these pitfalls in Design and holes in Verification. This paper introduces a Specman based mixed approach that gives the advantage of both Code coverage and Functional coverage. In general Code coverage is easy to setup but analysis of coverage reports is difficult and needs to be validated with respect to the design, where as in Functional coverage the implementation needs more effort and interaction with designer but coverage analysis is more friendly, also it is validated for the specific design (because the verification engineer is defining the coverage points). The approach introduced in this paper requires minimum interaction with designer and coverage reports are self-explanatory. Now onwards the term coverage means the coverage methodology that introduced in this paper.

2.0 Objective of Coverage Analysis

Objective of coverage analysis is to detect the following scenarios

- **False Triggering in State Registers:** All the possible binary combination of "state registers" (set of flip-flop used for storing current state) may not legal for a state machine (for example in a One-hot encoded state machine, if there is more than or less than one bit set at a time is illegal). The state registers switching to illegal values can treated as false triggering.

- **Missing States:** Find out that any state is missing from the verification process (state machine is not getting entered in certain states).

- **Illegal Transitions:** In transition coverage, it is identifying the possible legal two state transitions, if any other transitions occurs at any time can be treat as illegal transition. Transitions are identifying from bubble diagram, so a false transition leads to the mismatch between bubble diagram and RTL code.

- **Missing Transitions:** Being in any state, it can be specified the full set of possible previous sates (by observing the bubble diagram (schematic representation of FSM)). Transition coverage identifies all possible, legal two state transitions (switching from one state to other). This can define at the same coverage group of states.

- **Coverage for Functional Path:** In FSMs there are functionally important paths; designer can specify the functional paths. A functional path is a collection of state values in definite sequence. When DUT is working, FSM will be in repetition of these functional paths at any time.

- **Expression Coverage for next State Determining Combo Logic:** If the FSM contains complex logical conditions with priority encoding basis for next state determination, it is preferable to put cross coverage for the relevant inputs at that state. These inputs need to be sampled with respect to the specific state information.

Working clock can be the default sampling event for states and its transitions, for expression coverage inputs can be sampled with events corresponding to the specific states. Coverage definitions can further extended for dependency between different functional paths and also with the states of other interacting FSMs.

3.0 Implementation in Specman

3.1 Detection of False Triggering and Missing States

Current state of the FSM is a coverage item of enumerated data type, and in the coverage definition file all possible state values of current states will be defined in that enumerated type with the same parameterized names in HDL, so that the coverage reports can be viewed in terms parameterized state names rather than its numerical values. Value of the current state register is sampled at every active edge of the clock from HDL; it will be of type unsigned integer and requires a typecasting before assigning to the coverage item (because it is enumerated data type in coverage definition file). Any false triggering happens means that particular numerical value of the current state is missing in enumerated type definition, and it will issue error during typecasting.

Guideline: Put a coverage item (state) for current state, this can be of enumerated data type so that coverage result can be viewed in terms of state names in design, also it helps to detect false triggering by issuing error while typecasting an undefined numerical value to its enumerated type. Since state is a coverage item, missing states will be indicated as holes in state.

3.2 Detection of Illegal and Missing, Two State Transitions

All the possible two state transitions (transition from one state to another) can be identified from the bubble diagram. Put a transition coverage for the above mentioned state using illegal option for all non-specified transitions, so that the occurrence of all the legal two state transition will be displayed in transition coverage. Occurrence of any unspecified transition leads to “Dut error” due to the illegal option.

Guideline: Define the legal transitions by observing the bubble diagram that helps in bringing out the mismatch between RTL code and design (Assumption: the bubble diagram is as per the design and it will infer correct functionality).

3.3 Coverage for Functional Paths

Functional paths in design can be identified with the help of designer. Each path will be a definite sequence of states, so keep an event for each state and trace them with temporal expression. When the temporal expression succeeds emit the path event. For each functional path there will be a path event (occurrence of this event indicates the

occurrence of that functional path) and a path flag (this flag will set every time when the specified path event occurs also it will reset all other path flags).

Guideline: Declare the path flags as Boolean, occurrence of functional path (indicated by path event) can specified as TRUE, occurrence of a functional path make the other paths to FALSE. The coverage report comes with total number of occurrences with the contribution of each path.

3.4 Expression Coverage for Combo Logic

Certain states in FSM will be complex in nature because it needs to check many inputs for a state change, also more number of branching from a state can make it complex. It is advisable to have expression coverage for all the active inputs (set of inputs that can make a state change) while the state machine is in that state. Expression coverage of the active inputs tells about the occurrence of those inputs and its possible combinations.

Guideline: Identify the complex states (by looking the branches and the level combo logic involved in state change) in FSM; sample all the active inputs in those state while FSM is in that state, put a cross coverage for the sampled inputs.

4.0 Sample FSM and its Coverage Analysis

4.1 Sample FSM

Sample FSM is taken from the SNUG paper “Synthesis Friendly FSMs”, SNUG India 2003.

```
/*
 ----------------------------------------------------------------------
 -- Module Name       : onehot_moore_fsm5
 -- Encoding          : One-hot
 -- Implementation    : Moore
 -- States            : 5
 -- Outputs           : Registered
 ----------------------------------------------------------------------
*/


module  onehot_moore_fsm5
  (
  // Inputs
  clk_i, rst_i, in1, in2, in3, in4, in5, in6, in7, in8, in9,

  // Output
  out
  );

//--------------------- Global parameters Declarations ---------------

  // Binary states
  parameter       IDLE = 0,  // Idle state
```

```
                  S1   = 1,  // ......
                  S2   = 2,  // ....
                  S3   = 3,  // ...
                  S4   = 4;  // ..

  // One-hot states
  parameter [S4 : IDLE]     IDLE_S = 1 << IDLE,
                            S1_S   = 1 << S1,
                            S2_S   = 1 << S2,
                            S3_S   = 1 << S3,
                            S4_S   = 1 << S4;

//---------------------- Input Declarations ----------------------------

  input  clk_i, rst_i, in1, in2, in3, in4, in5, in6, in7, in8, in9;

//---------------------- Output Declarations ---------------------------

  output [S4 : IDLE] out;

//---------------------- Output Registers ------------------------------

  reg    [S4 : IDLE] out;

//---------------------- Internal Register Declarations ----------------

  reg    [S4 : IDLE] tmp_out;

//---------------------- State Registers -------------------------------

  reg    [S4 : IDLE] next_state, current_state;

//---------------------- Start of Code ---------------------------------


// Combinational part of FSM

  always @(in1 or in2 or in3 or in4 or in5 or in6 or
           in7 or in8 or in9 or current_state) begin

    case (1'b1)  // synopsys parallel_case

      current_state[IDLE] : begin  // State 1

        if (in1 && in2 && ~in3 && in4) begin
          next_state = S1_S;
        end
        else begin
          next_state = IDLE_S;
        end
      end

      current_state[S1] : begin  // State 2

        if (~in1 || ~in2 || in5) begin
          next_state = S4_S;
        end
        else begin
          if (in9) begin
            next_state = S2_S;
          end
          else begin
            next_state = S1_S;
          end
```

```
      end
    end

    current_state[S2] : begin // State 3

      if (in1 && in2) begin
        if (in6 && in7) begin
          next_state = S3_S;
        end
        else begin
          next_state = S2_S;
        end
      end
      else begin
        next_state   = S4_S;
      end
    end

    current_state[S3] : begin // State 4

      if (in1 && in2) begin
        next_state = S3_S;
      end
      else begin
        next_state = S4_S;
      end
    end

    current_state[S4] : begin // State 5

      if (in8) begin
        next_state = IDLE_S;
      end
      else begin
        next_state = S4_S;
      end
    end

    default : begin

      next_state = IDLE_S;

      // synopsys translate_off
      $display (" FSM is in invalid state, switching to IDLE ");
      // synopsys translate_on
    end
  endcase
end


//  Sequential part of FSM - Registering the outputs & state

  always @(posedge clk_i or negedge rst_i) begin

    if (~rst_i) begin
      out           <= 5'b0_0000;
      current_state <= IDLE_S;
    end
    else begin
      out           <= tmp_out;
      current_state <= next_state;
    end
  end
```

```
//  Output generation

  always @(current_state) begin

    case (1'b1)  // synopsys parallel_case

      current_state[S1] : tmp_out = 5'b0_0010;

      current_state[S2] : tmp_out = 5'b0_0110;

      current_state[S3] : tmp_out = 5'b0_1110;

      current_state[S4] : tmp_out = 5'b1_0000;

      default :           tmp_out = 5'b0_0001;
    endcase
  end

endmodule
```

Observations

States: One-hot (inputs from RTL code)
States: IDLE_S, S1_S, S2_S, S3_S and S4_S

Legal two state transitions: 12 (inputs from Bubble diagram)
Legal two state transitions: (state == IDLE_S and prev_state == IDLE_S)
(state == IDLE_S and prev_state == S4_S)
(state == S1_S and prev_state == S1_S)
(state == S1_S and prev_state == IDLE_S)
(state == S2_S and prev_state == S2_S)
(state == S2_S and prev_state == S1_S)
(state == S3_S and prev_state == S3_S)
(state == S3_S and prev_state == S2_S)
(state == S4_S and prev_state == S4_S)
(state == S4_S and prev_state == S1_S)
(state == S4_S and prev_state == S2_S)
(state == S4_S and prev_state == S3_S)

Functional paths: 3 (inputs from Designer)
Functional paths: path1 - IDLE_S > S1_S > S4_S > IDLE_S
path2 – IDLE_S > S1_S > S2_S > S4_S > IDLE_S
path3 – IDLE_S > S1_S > S2_S > S3_S > S4_S > IDLE_S

Complex States & its active inputs: 3 (inputs from RTL code)
Complex States & its active inputs: IDLE_S; in1, in2, in3, in4
S1_S; in1, in2, in5, in9
S2_S; in1, in2, in6, in7

4.2 Coverage Definition

```
-------------------------------------------------------------------------
-- Module Name       : u_onehot_moore_fsm5_cov.e
-- Function          : Coverage definitions for u_onehot_moore_fsm
-------------------------------------------------------------------------

<'

// Enumerated type definition for FSM states
type T_onehot_moore_fsm5 : [IDLE_S    = 5'b0_0001,
                            S1_S      = 5'b0_0010,
                            S2_S      = 5'b0_0100,
                            S3_S      = 5'b0_1000,
                            S4_S      = 5'b1_0000] (bits: 5);


// Instantiation of coverage unit in top, declare the coverage
// definition of each FSM in separate unit
extend u_e_top
{
  u_onehot_moore_fsm5_cover       : onehot_moore_fsm5_cov is instance;
  keep  u_onehot_moore_fsm5_cover.hdl_path() == "u_onehot_moore_fsm5";
};


// Coverage defining unit for u_onehot_moore_fsm5
unit onehot_moore_fsm5_cov
{

--------------------- Virtual Field Declaration ----------------------
// Functional paths
 !path1 : bool;
 !path2 : bool;
 !path3 : bool;
--------------------- Event Declaration ------------------------------

// Clock event
 event clk      is rise ('clk_i') @sim;

// State events
 event e_IDLE_S is
true('current_state'.as_a(T_onehot_moore_fsm5)==IDLE_S)@clk;
 event e_S1_S   is
true('current_state'.as_a(T_onehot_moore_fsm5)==S1_S)  @clk;
 event e_S2_S   is
true('current_state'.as_a(T_onehot_moore_fsm5)==S2_S)  @clk;
 event e_S3_S   is
true('current_state'.as_a(T_onehot_moore_fsm5)==S3_S)  @clk;
 event e_S4_S   is
true('current_state'.as_a(T_onehot_moore_fsm5)==S4_S)  @clk;

// Temporal expression for defining functional path with events
// More specific knowledge about the occurrence (number of clocks
// in which the FSM exists in a state) helps to write more
// specific expression.

 event e_path1 is {[..]*@e_IDLE_S;
                   @e_S1_S;   [..]*@e_S1_S;
                   @e_S4_S;   [..]*@e_S4_S;
                   @e_IDLE_S;} @clk;
```

```
 event e_path2 is {[..]*@e_IDLE_S;
                   @e_S1_S;   [..]*@e_S1_S;
                   @e_S2_S;   [..]*@e_S2_S;
                   @e_S4_S;   [..]*@e_S4_S;
                   @e_IDLE_S} @clk;


 event e_path3 is {[..]*@e_IDLE_S;
                   @e_S1_S;   [..]*@e_S1_S;
                   @e_S2_S;   [..]*@e_S2_S;
                   @e_S3_S;   [..]*@e_S3_S;
                   @e_S4_S;   [..]*@e_S4_S;
                   @e_IDLE_S} @clk;

// Sampling event for path coverage
 event e_path  is {@e_path1 or @e_path2 or @e_path3} @clk;

--------------------- On Struct Member -------------------------------

// Setting/Resetting functional path flags
 on e_path1
 {
   path1 = TRUE;
   path2 = FALSE;
   path3 = FALSE;
   out (sys.time, " e_path1 occurred ");
 };

 on e_path2
 {
   path1 = FALSE;
   path2 = TRUE;
   path3 = FALSE;
   out (sys.time, " e_path2 occurred ");
 };

 on e_path3
 {
   path1 = FALSE;
   path2 = FALSE;
   path3 = TRUE;
   out (sys.time, " e_path3 occurred ");
 };

--------------------- Coverage Groups --------------------------------

 cover clk  using text = "onehot_moore_fsm5 coverage" is {

    // State coverage
    item state :
T_onehot_moore_fsm5='current_state'.as_a(T_onehot_moore_fsm5);


    // 2 State Transition coverage
    transition state using text = "2 state transitions", illegal =
       not((state == IDLE_S  and  prev_state == IDLE_S)  or
           (state == IDLE_S  and  prev_state == S4_S)    or
           (state == S1_S    and  prev_state == S1_S)    or
           (state == S1_S    and  prev_state == IDLE_S)  or
           (state == S2_S    and  prev_state == S2_S)    or
           (state == S2_S    and  prev_state == S1_S)    or
           (state == S3_S    and  prev_state == S3_S)    or
           (state == S3_S    and  prev_state == S2_S)    or
           (state == S4_S    and  prev_state == S4_S)    or
```

```
            (state == S4_S    and  prev_state == S1_S)    or
            (state == S4_S    and  prev_state == S2_S)    or
            (state == S4_S    and  prev_state == S3_S));

  };


  // Coverage for functional paths
  cover e_path using text = "functional path coverage" is {

      item path1;
      item path2;
      item path3;
  };


  // Expression coverage for complex conditions in
  // the next state determining combinational logic
  cover e_IDLE_S using text = "expression coverage at IDLE_S state" is
{

      item in1 : bit = 'in1';
      item in2 : bit = 'in2';
      item in3 : bit = 'in3';
      item in4 : bit = 'in4';

    cross in1, in2, in3, in4;
  };

  cover e_S1_S using text = "expression coverage at S1_S state" is {

      item in1 : bit = 'in1';
      item in2 : bit = 'in2';
      item in5 : bit = 'in5';
      item in9 : bit = 'in9';

    cross in1, in2, in5, in9;
  };

  cover e_S2_S using text = "expression coverage at S2_S state" is {

      item in1 : bit = 'in1';
      item in2 : bit = 'in2';
      item in6 : bit = 'in6';
      item in7 : bit = 'in7';

    cross in1, in2, in6, in7;
  };

};  -- onehot_moore_fsm5_cov

'>
```

Coding Guidelines

- Declare the FSM states as enumerated data types with proper dimension and values. It is advisable to use the same parameterized HDL state names in enumerated types, so that coverage reports become more readable.

- Define a Specman event for clock (use @sim option) and all other event can be with respect to the Specman clock. This gives good runtime performance because there is only one event declaration with simulator call back.

- More specific knowledge about the functionality helps to define precise and accurate temporal expression for path coverage. The temporal expression showed in code is the de-facto one, suitable for any functionality.

- State and its transition coverage can be defined with respect to the Specman clock event, because current state change occurs synchronous with clock.

- Functional path coverage can be defined with respect to the consolidated path event (logically ORed version of all path events).

- Functional path flags can declared as Boolean and functional path occurrence can indicated by TRUE, FALSE will indicate a non-occurrence.

- Cross coverage of the combinational inputs can sampled with respect to the corresponding state events.

- Interaction with the designer is required for identifying functional paths; all other coverage points could be extracted from bubble diagram and RTL code.

- It is advisable to define the transition coverage by analyzing the bubble diagram. It helps in bringing out the mismatch between design and RTL code. If the RTL code misses any transition that will indicate as hole, where as design miss a transition and if it occurs in code that will hit illegal condition (hence dut error).

- If there is any dependency exists between functional paths, it can be monitored with temporal expressions.

The coverage methodology introduced in this paper not checking the mere occurrence of conditions and states, it is ensuring the occurrence and trying to map it with useful features of the design. Coverage definitions and temporal expressions are act like monitors. It makes possible to get the advantage of both Code coverage and Functional coverage and the coverage results can club with functionality. A high coverage result can assure good amount of functional correctness in implementation.

4.3 Coverage Report Analysis

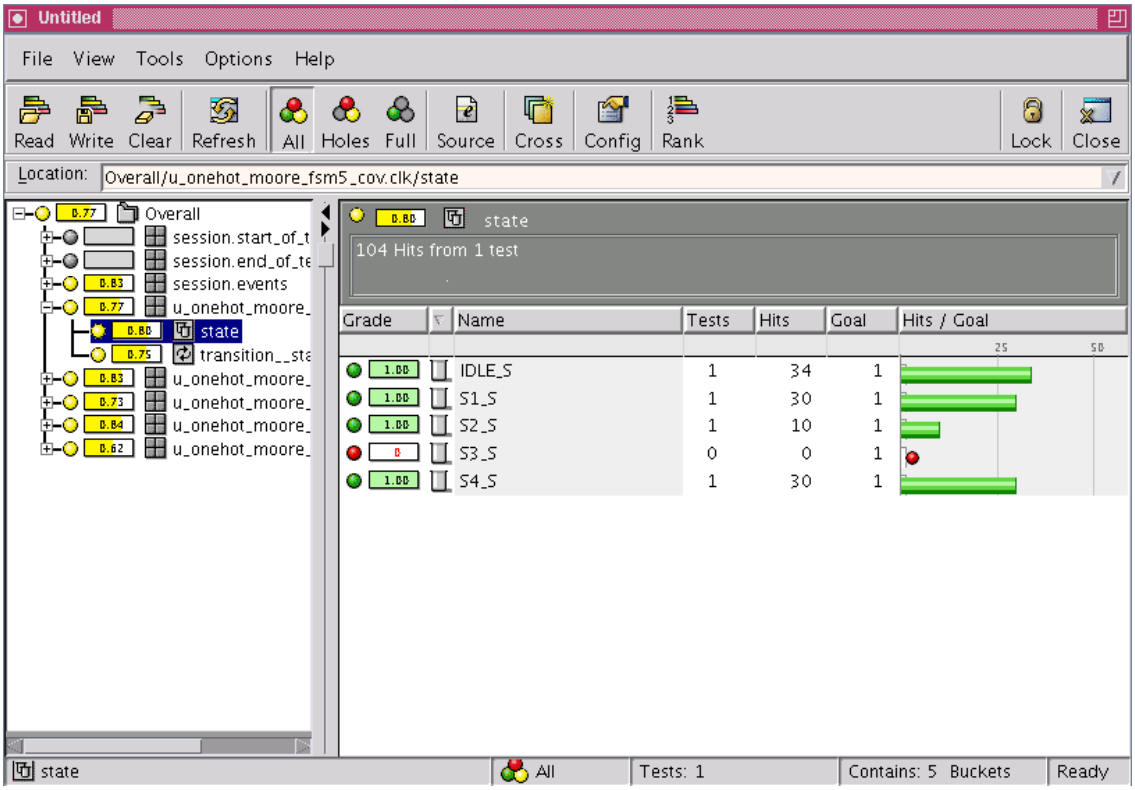

Fig 1: State Coverage

Missing state: S3_S

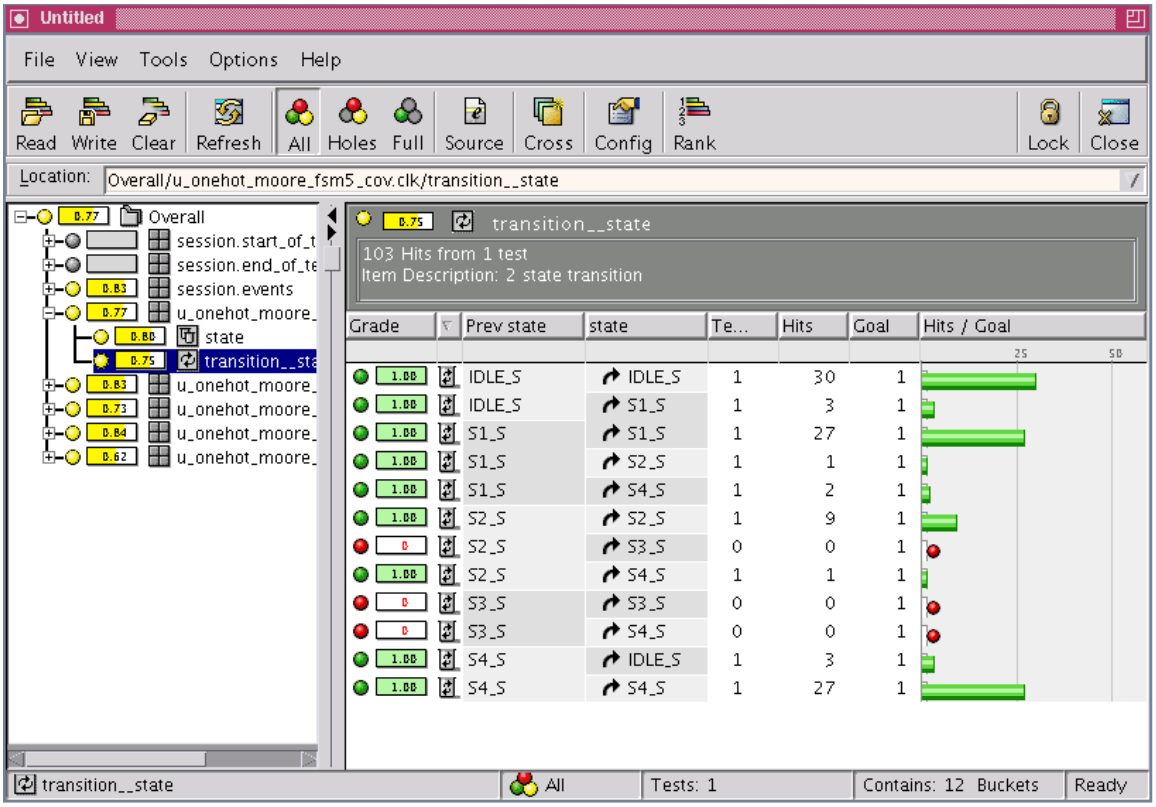

Fig 2: Two State Transition Coverage

Missing transitions are S2_S > S3_S,
S3_S > S3_S,
S3_S > S4_S.

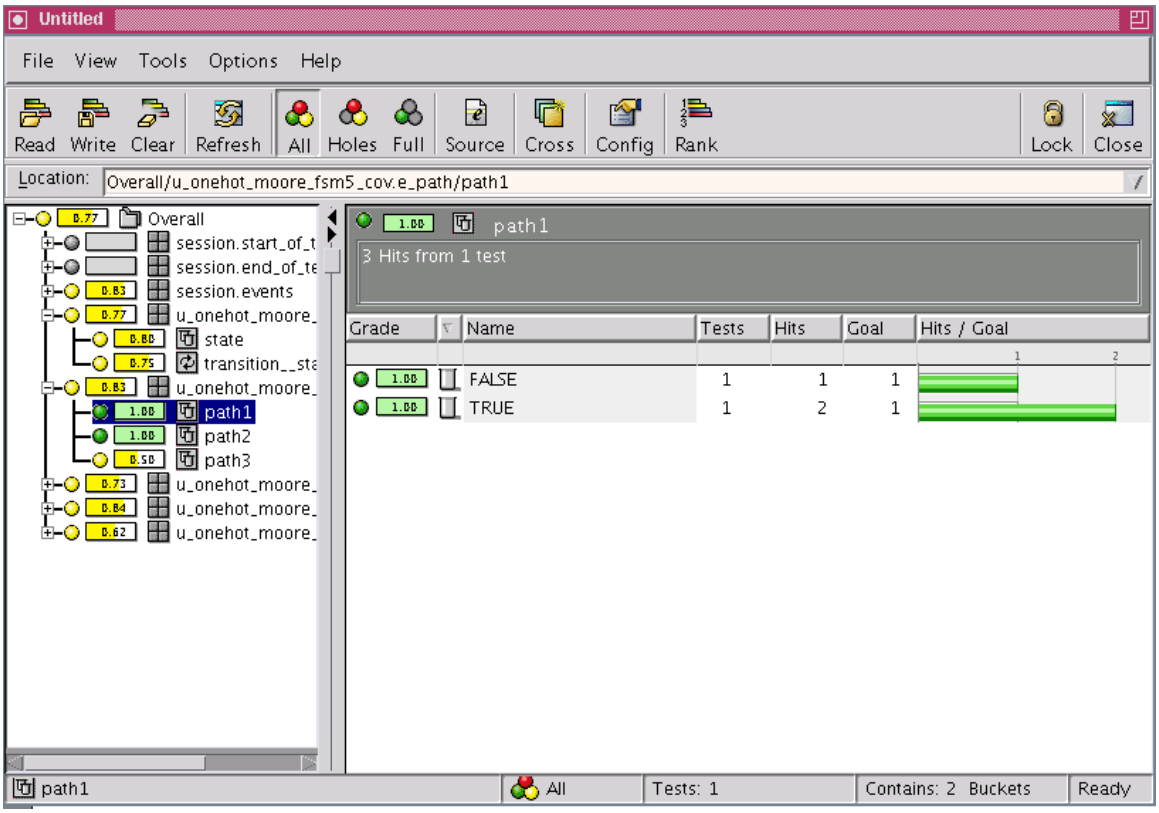

Fig 3: path1 Coverage

Total occurrence of functional paths: 1+ 2 = 3

Occurrence of functional path1: 2

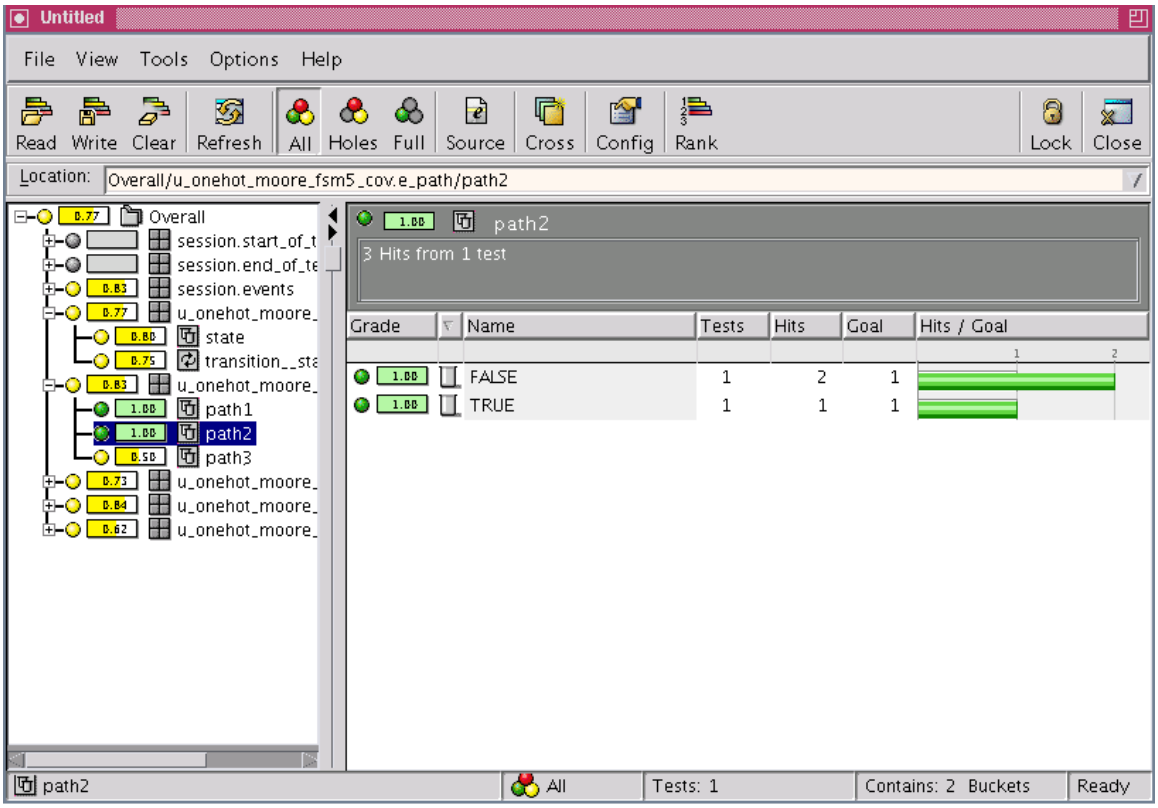

Fig 4: path2 Coverage

Total occurrence of functional paths: 2 +1 = 3

Occurrence of functional path2: 1

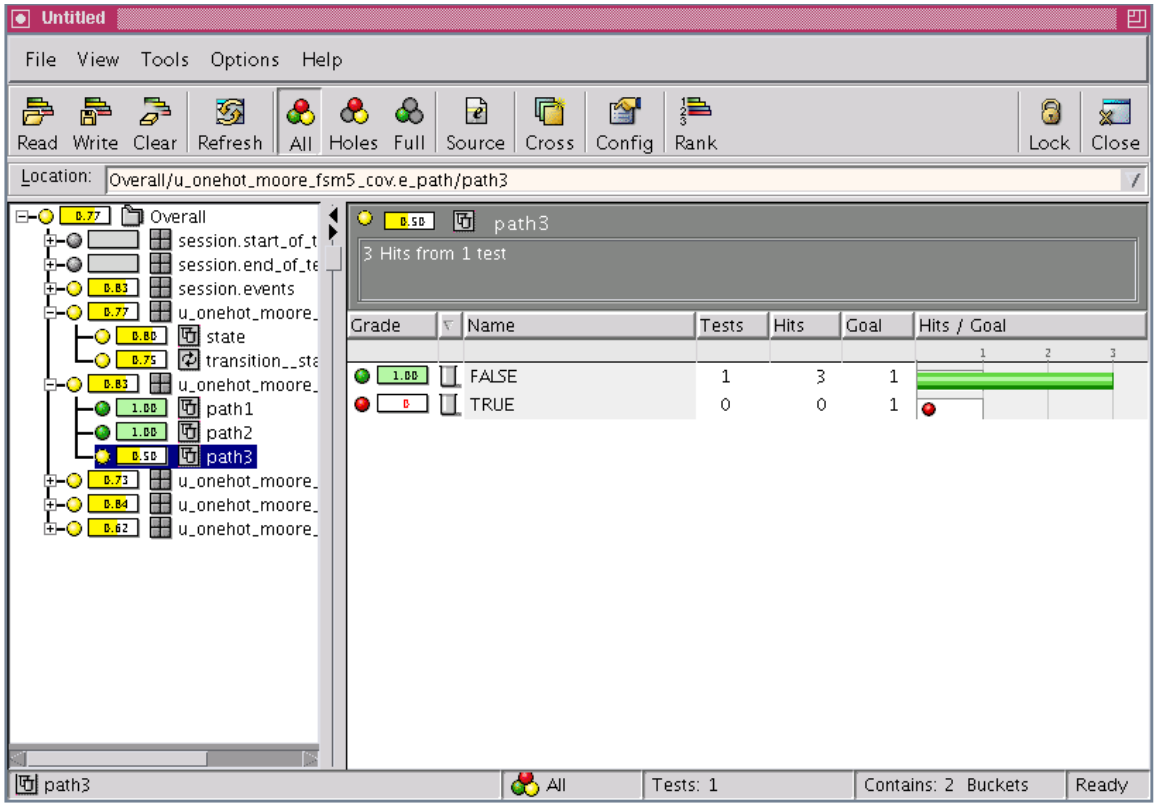

Fig 5: path3 Coverage

Total occurrence of functional paths: 3

Occurrence of functional path3: 0

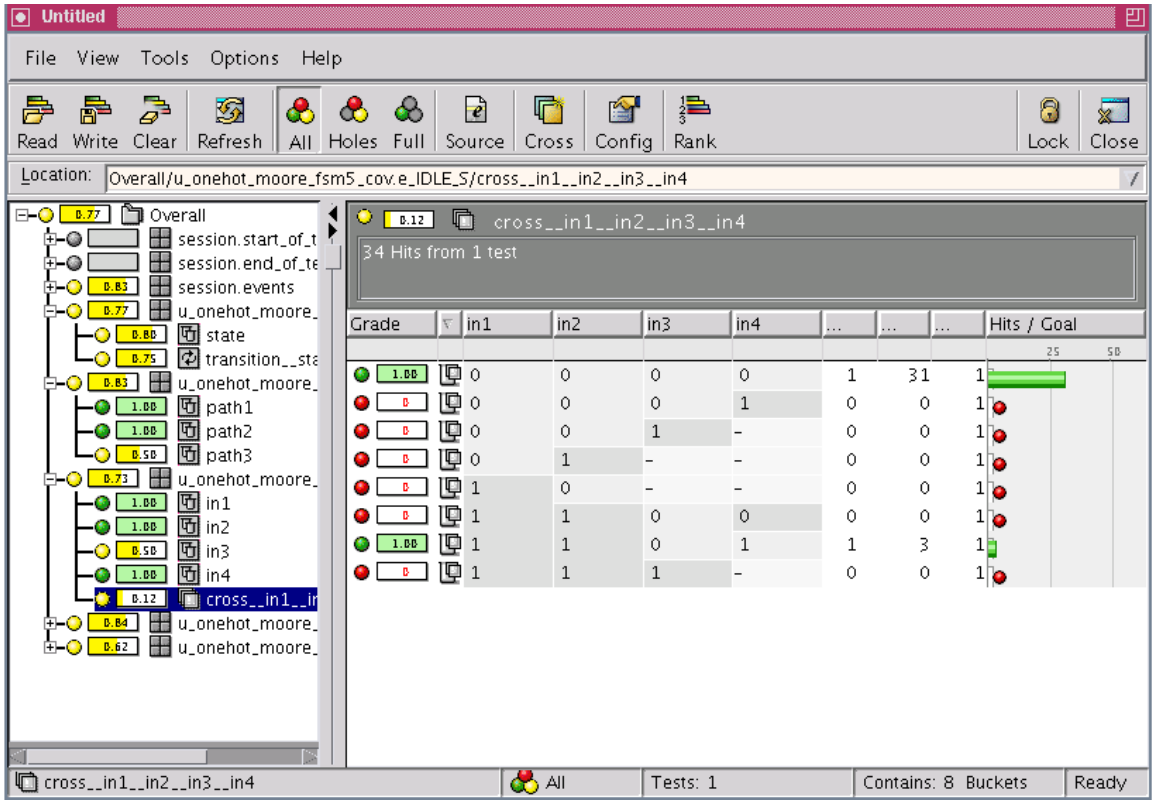

Fig 6: Cross Coverage for Inputs in IDLE_S State

Active inputs combinations that hit at IDLE_S (in1, in2, in3, in4): (0,0,0,0), (1,1,0,1).

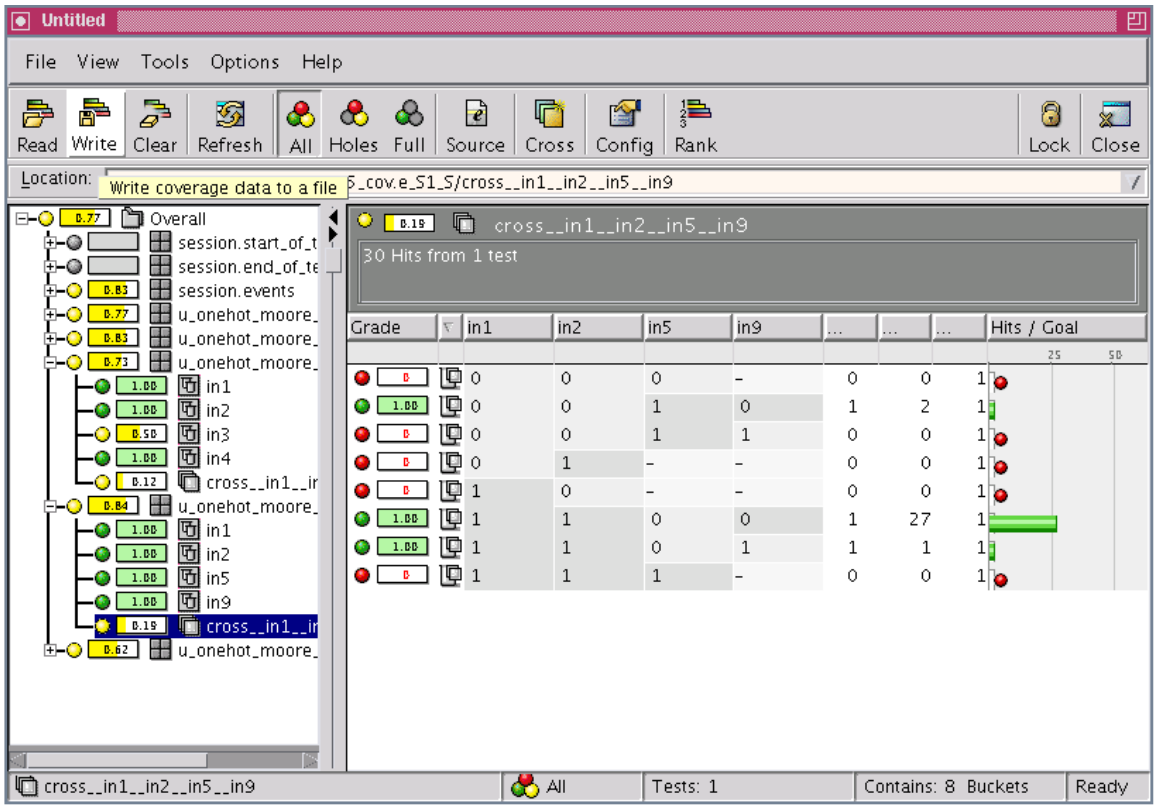

Fig 7: Cross Coverage for Inputs in S1_S State

Active inputs combinations that hit at S1_S (in1, in2, in5, in9): (0,0,1,0),
(1,1,0,0),
(1,1,0,1).

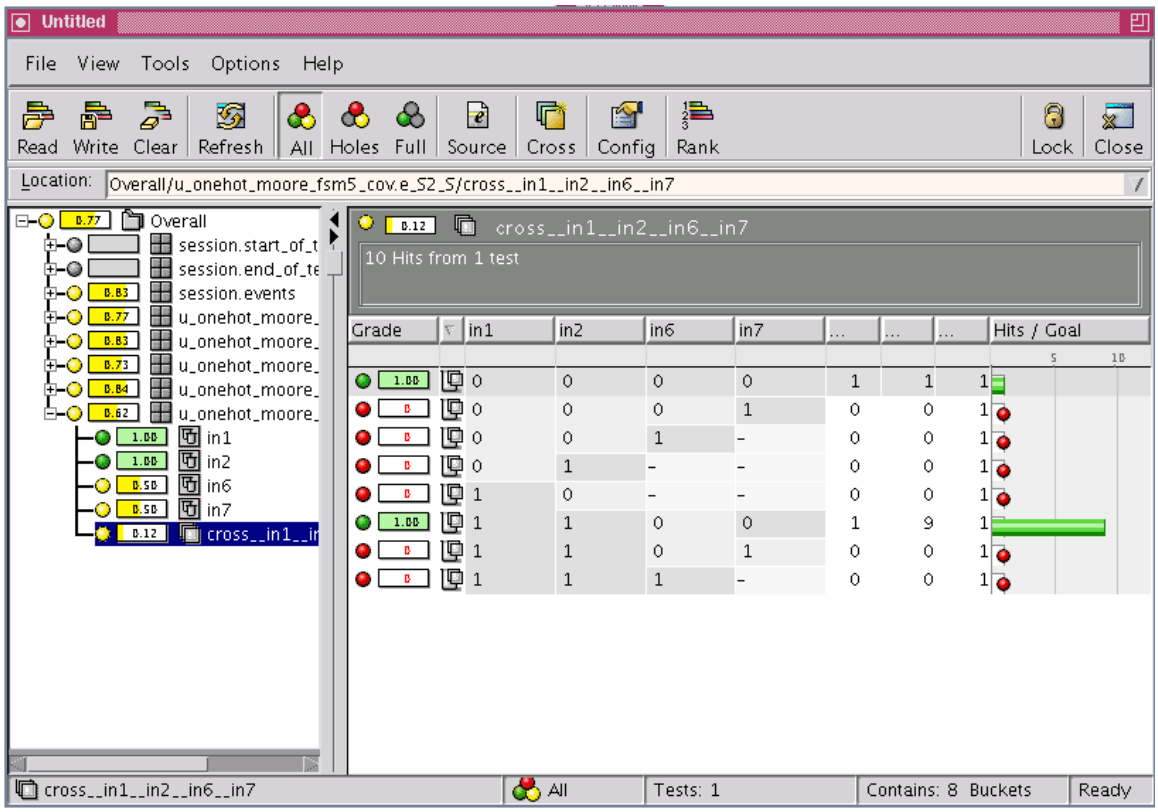

Fig 8: Cross Coverage for Inputs in S2_S State

Active inputs combinations that hit at S2_S (in1, in2, in6, in7): (0,0,0,0), (1,1,0,0).

5.0 Acknowledgements

I would like to thank every one who is giving constant support and encouragement in my efforts.

6.0 References

- Specman Elite, “e Language Reference” Version 4.2
- Specman Elite, “Usage and Concepts Guide” Version 4.2
- Jacob Joseph, “Synthesis Friendly FSMs” SNUG India 2003.