Using SVA for scoreboarding and testbench design

Ben Cohen http://systemverilog.us/

Abstract

Though assertions are typically used for the verification of properties, they can be applied in many other verification applications. For example, in scoreboarding functions can be called from within a sequence match item after the assertion reaches a desired point.

Concept

If a sequence expression succeeds, an attached sequence match item would be executed if it exists. Per 1800, Annex A, a *sequence_match_item* can be one of the following:

operator_assignment
| *inc_or_dec_expression*
| *subroutine_call*

The access to a subroutine call from a sequence match item is the unique feature that allows many options in the construction of the testbench. For example, the subroutine call can do one (or more) of the following:

1. covergroup sampling, as shown below
(http://systemverilog.us/papers/cpu_cg_module.sv)

```
sequence q_fetch;  @ (clk)
  mode==FETCH ##1 (1, instr_cg1.sample(), addr_cg1.sample());
endsequence : q_fetch
cover property (q_fetch);
```

2. Calculating the successful ranges in which an assertion was exercised.
(http://systemverilog.us/papers/reqack_sequence_cover.sv)

```
int reqackdone; // count
default clocking cb_ck1 @ (posedge clk);  endclocking

// count
function void set_reqackdone(int v);
    reqackdone=v;
endfunction : set_reqackdone

property p_seq_reqack;
    int v_count;
    (req, v_count=0)
       ##1 (rdy, v_count+=1)
       ##1 (1, v_count+=1)[*1:3]
       ##0 (done, set_reqackdone(v_count), cg_inst.sample());
endproperty : p_seq_reqack
// cp_seq_reqack: cover property(req ##1 rdy ##[1:3] done);
cp_seq_reqackP: cover property(p_seq_reqack); // <------------

covergroup cg_reqack_path_length;
    reqackdone_cp : coverpoint reqackdone
    {
        bins a2 = { 2};
```

```
            bins a3 = { 3 }; // { [3:3] };
            bins a4 = { 4 }; // { [4:4] };
        }
    endgroup
    cg_reqack_path_length cg_inst = new();
```

3. Performing complex scoreboarding operations at specific points in the assertion. The assertion can be brought to a specific FSM point, and then call functions to do the scoreboarding. The next subsection addresses this application by example.

Application example

In the following example (http://systemverilog.us/papers/matrix.sv http://systemverilog.us/papers/matrix2.sv) the model holds an image in a 25x25 matrix array. Each pixel in that image is an unsigned integer. The image is partitioned into 25 slices or quadrants, and the scoreboard needs to compute within a 5x5 slice the number of pixels greater than 3. The image is loaded when `done_image` is true, and the slice to be selected is determined at a new `ld` signal. The `done_image` occurs within 1 to 3 cycles after the `ld` signal.

In this model, the selection of the slice to be processed is randomly selected; this approach demonstrates the application of function calls from a sequence match item. Two functions are called from the sequence match item of a sequence in a property. These include:

```
module matrix;
   int unsigned slice [0:4][0:4];
   int unsigned image [0:24][0:24];
   bit clk;
   bit ld, done_image;
   int count;
   typedef struct {
       int x;
       int y;
   } quadrand_t;
   // quadrand_t qd={0,1};
   quadrand_t found_qd;


   // Function select_slice copies pixels from an image into a slice
   // It is used by the count_targets_in_quadrant function.
   // There are 25 quadrants in the image, ranging from quadrant 0 to quadrant 24.
   function void select_slice(quadrand_t qd);
       int unsigned v [0:4][0:4];
       for (int i=0; i<5; i++) begin
           for (int j=0; j<5; j++) begin
               v[i][j]=image[qd.x*4+i][qd.y*4+j];
           end
       end
       slice=v;
   endfunction : select_slice
```

```
// Function find_quadrant randomly finds a quadrant
// Called by property
function quadrand_t find_quadrant();
    logic[0:3] x, y;
    quadrand_t v_qd;
    if (!randomize(x, y) with {x <5; y<5;})
        `uvm_error("MYERR", "This is a randomize error of find_quadrant");
    v_qd.x=x;
    v_qd.y=y;
    found_qd = v_qd;
    return v_qd;
endfunction : find_quadrant

// Function count_targets_in_quadrant counts the
// number pixels >3 within the selected slice
// Called by property
function void count_targets_in_quadrant(quadrand_t v_qdt);
    automatic int unsigned v_count=0;
    select_slice(v_qdt);
    foreach (slice[i,j]) begin
        if (slice[i][j] > 3) v_count++; // 3 is a threshold
    end
    count = v_count;
endfunction : count_targets_in_quadrant
```

When the assertion detects a new `ld`, the local variable `v_qdt` of the property saves value of the selected quadrant through a call to `find_quadrant`. When `done_image` is detected, the number of pixels > 3 within that selected quadrant is computed through a call to `count_targets_in_quadrant(v_qdt)` with the previously computed quadrant passed as an actual argument.

```
default clocking cb_ck1 @ (posedge clk);  endclocking
property p_targets;
    int unsigned v_slice [0:4][0:4];
    quadrand_t v_qdt;
    first_match(($rose(ld), v_qdt=find_quadrant()) ##[1:3] done_image) |->
       (1, count_targets_in_quadrant(v_qdt));
endproperty : p_targets

ap_targets: assert property(p_targets);
```

Where should assertion be written?

Concurrent assertions are illegal in classes. Thus, assertions are best written in modules, or modules bound to other modules, or in interfaces. The body functions may use class members or methods. However, assertions may not use members referenced with class handles. Thus, the following is illegal:

```
class c;
      int unsigned slice [0:4][0:4];
    int unsigned image [0:24][0:24];
    typedef struct {
        int x;
```

```
        int y;
    } quadrand_t;

    quadrand_t found_qd;
    int count;

    function logic[0:3] find_quadrant(bit v);
        static logic[0:3] x, y;
        //automatic quadrand_t v_qd;
        if(v) begin
          if (!randomize(x, y) with {x <5; y<5;})
            `uvm_error("MYERR", "This is a randomize error of find_quadrant");
            return x;
        end
        else return y;
    endfunction : find_quadrant
..
endclass : c
module matrix;
    parameter seed=10;
    c c_h=new();
    property p_targets;
        int unsigned v_slice [0:4][0:4];
        logic[0:3] v_x, v_y;
        first_match(($rose(ld), v_x=c.h_find_quadrant(1),  // illegal referencing
                                v_y=c_h.find_quadrant(0)) ##[1:3] done_image)
        |->
          (1, c_h.count_targets_in_quadrant(v_x, v_y));
    endproperty : p_targets
```