Solving Complex Users' Assertions
by Ben Cohen

Abstract:
The verificationacademy.com/forums/ is an interesting interactive SystemVerilog forum where users seek solutions to real application issues/problems. Many of those questions are about assertions, and SVA has very specific set of rules that do not necessarily address complex users' requirements. This paper brings a collection of a few most interesting and challenging users’ questions and provide solutions along with explanations about getting around (or working with) SVA, or using other alternatives (see my paper SVA Alternative for Complex Assertions). It turns out that many of these solutions require a different point of view in approaching the assertions, and often require supporting logic. All code along with simple testbenches is provided.

1.1 Dynamic delays and repeats

ISSUE: Using dynamic values for delays or repeats is illegal in SVA; how can this be easily resolved?

```verilog
int dly1=2, dly2=7; // module variables
ap_abc_delay: assert property($rose(a) #dly1 b |-> #dly2 c); // ILLEGAL SVA
ap_abc_repeat: assert property($rose(a) |-> b[*dly1] #1 c); // ILLEGAL SVA
```

SOLUTION: Reference ii (at end of this paper) provides a solution for handling dynamic delays an repeats using tasks. However, in the verificationacademy.com/forums/systemverilog forum, a user brought up a very interesting alternative that uses a package; it is presented here. The concept is very simple, the repeat or delay sequence is saved in a package with two defined sequence declarations that include arguments.

```
http://SystemVerilog.us/vf/sva_delay_repeat_pkg.sv
package sva_delay_repeat_pkg;
sequence dynamic_repeat(q_s, count);
  int v=count;
  (1, v=count) ##0 first_match((q_s, v=v-1'b1) [*1:$] ##0 v<=0);
endsequence

sequence dynamic_delay(count);
  int v;
  (1, v=count) ##0 first_match((1, v=v-1'b1) [*0:$] ##1 v<=0);
endsequence
endpackage
```

The package can be applied as follows:

```
http://SystemVerilog.us/vf/sva_delay_repeat.sv
import sva_delay_repeat_pkg::*;
module top;
  timeunit 1ns; timeprecision 100ps;
  bit clk, a, b, c=1;
  int r=2;
  default clocking @(posedge clk); endclocking
  sequence q1; a ##1 b; endsequence

  ap_abr: assert property(a |-> dynamic_repeat(q1, r) #1 c);
  ap_delay: assert property(a |-> dynamic_delay(r) ##0 b);
```

1.2 No 2nd successful attempt before completion of first attempt; 2nd attempt is a fail

**ISSUE:** This was a difficult set of requirement to express. If 2 consecutive *req* and then one *ack*, the *ack* is for the first *req* attempt and that assertion passes. However, the 2nd *req* attempt causes that 2nd assertion to fail, regardless of the received *ack*. The following solution (assuming a default clocking) fails to work because all successful attempts of *req* can be satisfied by one *ack*, provided then meet the delay constraints.

\[
\text{\$rose}(\text{req}) \rightarrow \#\![1:10] \text{ ack}; \quad // \text{DOES NOT MEET THE REQUIREMENTS.}
\]

**SOLUTION:** To solve this conflict, there is a need to distinguish a real first *req* attempt from other secondary attempts. This can be accomplished with 1) the use of a *function* and a module *tag* bit. The *tag* bit is a flag that when *flag*=1 identifies that a first *req* was already initiated. The function checks the *tag* and returns *zero* if set. Otherwise, when *flag*=0, the function sets it to ONE and returns ONE, meaning that this is a first occurrence of *req*. The property uses a local variable called *go*; that local variable enables the property to continue checking for an *ack*, or immediately fail if it is zero. The *tag* bit is reset upon an assertion pass. In this case, if the first assertion fails, the *tag* bit never gets reset and all further assertions will fail (unless some external support logic resets the *flag* bit).

http://SystemVerilog.us/fv/reqack_special.sv
http://SystemVerilog.us/fv/reqack_special.png

```verilog
bit clk, req, ack, tag;
default clocking @(posedge clk); endclocking
function bit check_tag();
  if(tag) return 1'b0;
  else begin
    tag=1'b1;
    return 1'b1;
  end
endfunction

function void reset_tag();
  tag =1'b0;
endfunction
property reqack;
  bit go;
  @(posedge clk) ($rose(req), go=check_tag()) ->
  go #\![1:10] (ack, reset_tag()); // locks all future req if assertion fails
endproperty
ap_reqack: assert property(reqack);
```

1.3 Each successful attempt has its own exclusive completion consequent

**ISSUE:** This is a variation to the previous requirements; in this case, each *req* attempt is terminated with its own individual *ack*.

**SOLUTION:** To accomplish this, one could use concepts of a familiar model seen in hardware stores, typically in the paint department. There, the store provides a spool of tickets, each with a number. As a customer comes in, each customer takes a ticket. The clerk serving the customers has a sign that reads "NOW SERVING, TICKET #X". The customer that has the ticket gets served, the others have to wait. When done, the number X in incremented, and the next in-line customer gets served.
To solve this in SVA, one could use two variables: ticket, now_serving. A function is used to increment the ticket number, and a pass or a fail of the assertion increments the now_serving. The assertion code could then be written as follows:

```
bit clk, req, ack;
int ticket, now_serving;

function void inc_ticket();
    ticket = ticket + 1'b1;
endfunction

property reqack_unique;
    int v_serving_ticket;
    @(posedge clk) ($rose(req), v_serving_ticket=ticket, inc_ticket()) |->
        ##[1:10] now_serving==v_serving_ticket ##0 ack;
endproperty

ap_reqack_unique: assert property(reqack_unique)
    now_serving = now_serving+1; else now_serving = now_serving+1;
```

1.4 Every wr has a enb; no enb if no pending wr

**ISSUE:** There can be consecutive wr commands; every wr has an enb. If there is an enb with no pending wr, then it is an error. Thus,

\[
\begin{align*}
\text{wr} & \quad \text{wr} \quad \text{enb} \quad \text{... wr ... enb ... enb} \quad \text{// LEGAL} \\
+ & \quad + \quad + \\
\text{wr} & \quad \text{wr} \quad \text{enb} \quad \text{... enb ... enb}^* \quad \text{// ** ERROR, no prior wr for enb}
\end{align*}
\]

**SOLUTION:** The easiest solution for this type of requirement is to just use supporting logic; a counter is incremented for each wr occurrence, and decremented for each enb occurrence. An immediate assertion tests that the value of the counter is always greater than zero, or is zero.

```
int counter=0;
always @(posedge clk) begin
    if(wr && enb) ; // no change
    else if(wr) counter <= counter +1'b1;
    else if(enb) counter <= counter -1'b1;
    ap_wrrd: assert(counter >= 0);
end
```
1.5 Activate array of assertions based of dynamically-defined size

**ISSUE:** The design incorporates a dynamic req/ack signal pairs (logic[0:3] req, ack). Assertions are based on a sized set of pairs, and that size is dynamically set during runtime (bit[1:0] size=3) in the configuration phase. Thus, what is desired is something like the following:

```systemverilog
logic[0:3] req, ack;
bit[1:0] size=3;
property req_with_ack(logic req, logic ack); // @(posedge clk) disable iff (!reset)
   $rose(req) |=> $rose(ack);
endproperty
always @(posedge clk) begin
   for (int i=0; i<size; i++) begin
      ap_i: assert property(req_with_ack(req[i], ack[i]));
   end
end
```

Since assertions are statically allocated during elaboration, the above assertions will not compile.

**SOLUTION:** As a result of this restriction, one solution is to use the task approach described in SVA Alternative for Complex Assertions (see ref ii). Below is that solution:

```systemverilog
http://SystemVerilog.us/fv/reqack_dyn.sv
bit clk, reset=1'b1;
logic[0:3] req, ack, req_past, ack_past;
bit[1:0] size=3;
event e0, e; // for debug
task automatic t_req_with_ack(logic req, logic ack);
   if (!reset) return;
   if(req && !req_past) begin : rose // $rose(req) is illegal here
      -> e0; // automatic variables cannot be used in 'past'
      @posedge clk);
      a_reqack: assert (ack && !ack_past);
      -> e;
      return;
   end : rose
else return; // optional here
endtask
always @(posedge clk) begin
   for (int i=0; i<size; i++) begin
      fork
         t_req_with_ack(req[i], ack[i]);
      join_none
   end
end
```

Bad handle or reference
1.6 Sig "a" shall change values "n" times between sig "b" and sig "c"

**ISSUE:** Signal a changes n times between signal b and c. The value of n is static, but it could be dynamic.

**SOLUTION:** If n is static, the solution is rather simple. This solution makes use of SVA operators.

```systemverilog
import uvm_pkg::*; `include "uvm_macros.svh" http://SystemVerilog.us/fv/a_n_bc.sv
module top;
    timeunit ins; timeprecision 100ps;
    bit clk, a, b, c;
    int k=4;
    event e0, e1; // for debug

    // Sig "a" shall change values "k" times between sig "b" and sig "c".
    task t_abcdyn(int vk);
        automatic int count;
        automatic bit v_a;
        v_a=a; // save current value
        -> e0;
        forever begin
            @(posedge clk);
            if(a != v_a) begin: changed
                if (count==vk & & !c) begin
                    `uvm_error("MYERR", $sformatf("%m : at %t Reached %d, before c, expected %d, got %d", $realtime, vk, count));
                    -> e1;
                    return;
                end : changed
                else begin : keep_count
                    count = count+1'b1;
                    v_a=a; // save current value
                end : keep_count
            end : toend
            if(c) begin : toend
                a_k: assert(count==vk) else
                    `uvm_error("MYERR", $sformatf("%m : at %t error in changed, expected %d, got %d", $realtime, vk, count));
                    -> e1;
                    return;
            end : toend
        endtask // forever

    ap_abck: assert property ($rose(b) |> (1, t_abcdyn(k)));
```

If n is dynamic, then the use of tasks is recommended. It's a bit complicated though! Below is code for Sig "a" shall change values "k" times between sig "b" and sig "c".

**Note:** The use of local variables as counter in a property would fail to work because the local variable written within the $changed(a) thread could not be read in the c [->1] thread because of the intersect operator. From 1800, the values assigned to the local variable before and during the evaluation of the composite sequence are not always allowed to be visible after the evaluation of the composite sequence.
1.7 Assertion Controls

**ISSUE:** Is there a way to link an assumption to specific assertions (or to disable an assumption for specific assertions)?

**SOLUTION:** SV1800'2017: 20.12 Assertion control system tasks describes the Assertion control syntax

```systemverilog
assert_control_task ::= 
    assert_task [ ( levels [ , list_of_scopes_or_assertions ] ) ] ;
| assert_action_task [ ( levels [ , list_of_scopes_or_assertions ] ) ] ;
| $assertcontrol ( control_type [ , [ assertion_type ] [ , [ directive_type ] [ , [ levels ] [ , list_of_scopes_or_assertions ] ] ] ] ) ;

assert_task ::= 
    $asserton 
| $assertoff 
| $assertkill
assert_action_task ::= 
    $assertpasson 
| $assertpassoff 
| $assertfailon 
| $assertfailoff 
| $assertnonvacuouson 
| $assertvacuousoff
list_of_scopes_or_assertions ::= 
    scope_or_assertion { , scope_or_assertion }
scope_or_assertion ::= 
    hierarchical_identifier
```

The `$assertcontrol` provides finer granularity in how and which types of assertions are controlled. The most readable way to express the values of the `control_type`, `assertion_type`, and `directive_type` fields is to use a package where those values are clearly defined as constants with the `let` directive.

```systemverilog
import uvm_pkg::*; `include "uvm_macros.svh"
package asncntrl_pkg; // http://SystemVerilog.us/fv/asncntrl_pkg.sv

// Control type
let LOCK = 1; // assertion control type
let UNLOCK = 2; // assertion control type
let ON = 3; // assertion control type
let OFF = 4; // assertion control type
let KILL = 5; // assertion control type
let PASSON = 6; // assertion control type
let PASSOFF = 7; // assertion control type
let FAILON = 8; // assertion control type
let FAILOFF = 9; // assertion control type
let NONVACUOUSON = 10; // assertion control type
let VACUOUSOFF = 11; // assertion control type

// Assertion types
let CONCURRENT = 1; // assertion_type, concurrent
let S_IMMEDIATE = 2; // assertion_type, simple immediate
let D_IMMEDIATE = 12; // assertion_type, Final and Observed deferred immediate
let ALL_ASSERTS = 15; // CONCURRENT|S_IMMEDIATE|D_IMMEDIATE
let EXPECT = 16; // assertion_type, expect
let UNIQUE = 32; // unique if and case violation
let UNIQUE0 = 64; // unique0 if and case violation
let PRIORITY = 128; // priority if and case violation

// Assertion directives
let ASSERT = 1; // directive_type for assertion control tasks
let COVER = 2; // directive_type for assertion control tasks
```
let ASSUME = 4; // directive_type for assertion control tasks
let ALL_DIRECTIVES = 7; // (ASSERT|COVER|ASSUME);
endpackage
module top;
import asncntrl_pkg::*;
bit clk, a, b;
event start_sim;
int count1=0, count2=0;
initial forever #5 clk=!clk;
property Px1;
   a |=> b;
endproperty

# $assertcontrol
ap_test_kill: assert property(@(posedge clk) a |=> 1) $assertkill(0, top.ap_test_kill);
ap_test_kill0: assert property(@(posedge clk) a |=> 1) $assertcontrol(KILL, ALL_ASSERTS, ALL_DIRECTIVES, 0, top.ap_test_kill0);
ap_test_off: assert property(@(posedge clk) a |=> 1) $assertoff(0, top.ap_test_off);
// $assertoff([levels[, list]]) is equivalent to
// $assertcontrol(OFF, ALL_ASSERTS, ALL_DIRECTIVES, levels [,list])
ap_x1: assert property(@(posedge clk) Px1) count1 <= count1 + 1'b1;
ap_x2: assert property(@(posedge clk) Px1) count2 <= count2 + 1'b1;
initial begin
   $assertcontrol(KILL); // Stop checking all assertions
   wait (start_sim); // wait for subsystem to be ready to start checking for assertions
   $assertcontrol(ON); // enable all assertions
   // disable all pass action blocks except those needed
   $assertcontrol(asncntrl_pkg.LOCK, ALL_ASSERTS, ALL_DIRECTIVES, 0, ap_x1);
   // lock any changes to ap_x1
   $assertcontrol(PASSOFF); // pass off for ap_x2
   $assertcontrol(asncntrl_pkg.UNLOCK, ALL_ASSERTS, ALL_DIRECTIVES, 0, ap_x1, ap_x2);
end
// ...
endmodule : top

1.8 $past in SV Assertions

ISSUE: When signal a rises, check that busy was asserted sometime before (any number of clock cycles earlier). The following assertion fails because an infinite range in $past is not supported by SVA.
ap_INCORRECT: assert property(@(negedge clk)($rose(a) | - $past(busy,[1:$])))

SOLUTION: Instead of a looking-back approach with the $past, a forward looking approach solves this issue. Specifically, set a latch (ambusy) when busy is set. Reset the latch (ambusy) when the assertion either passes or fails. For example: http://SystemVerilog.us/fv/wasbusy.sv

bit clk,busy, ambusy, a;
always @(posedge clk) if(busy) ambusy <= 1'b1; // latch to 1
ap_a_was_bysy: assert property(@(negedge clk)$rose(a)|-> ambusy)
               ambusy=1'b0; else ambusy=1'b0; // resets to 0
1.9 Check clock period within tolerances

**ISSUE:** Check clock period within tolerances based on the `en` signal.

**SOLUTION:** Use local variable of type `realtime` in the property.

http://SystemVerilog/us/vf/ check_clk.sv

```systemverilog
timeunit 1ns; timeprecision 100ps;
bit clk, en, rst=0, a, b;
default clocking @(posedge clk); endclocking
realtime clk_period=20ns, clk_period_1=22ns, error_clk=2ns;

property p_period_enf;
  realtime current_time;
  disable iff (rst)
    (!en, current_time = $realtime) |=>
    (($realtime - current_time) <= (clk_period + error_clk))
    && (($realtime - current_time) >= (clk_period - error_clk));
endproperty : p_period_enf

ap_period_enf: assert property(p_period_enf);

property p_period_en;
  realtime current_time;
  disable iff (rst)
    (en, current_time = $realtime) |=>
    (($realtime - current_time) <= (clk_period_1 + error_clk))
    && (($realtime - current_time) >= (clk_period_1 - error_clk));
endproperty : p_period_en

ap_period_en: assert property(p_period_en);
```

1.10 Conclusions

Assertions have specific rules. Some requirements require the need for supporting logic along with module variables modified in the `sequence_match_item` with functions calls that may have side effects. Other occasions are best handled by tasks, as described in ref ii.

---

ben@SystemVerilog.us

\(^2\) SVA Alternative for Complex Assertions