Understanding the SVA Engine

Ben Cohen[i]

Abstract: Understanding the engine behind SVA provides not only a better appreciation and limitations of SVA, but in some situations provide features that cannot be simply implemented with the current definition of SVA. This paper first explains, by example, how a relatively simple assertion example can be written without SVA with the use of SystemVerilog tasks; this provides the basis for understanding the concepts of multithreading and exit of threads upon a condition, such as an error in the assertion. The paper then provides examples that uses computational variables within threads; those variables can cause, in some cases, errors in SVA. The strictly emulation model with tasks solves this issue.

1. Emulating a simple assertion:

With module variables "`a, b, c`" and a **`default clocking`**, consider the following SVA assertion:

```
ap_ab_then_c : assert property($rose(a) ##2 b |->  ##3 c);
```

This assertion can be emulating with an **`automatic`** task that is started from an **`always`** block. This firing of the task every `(posedge clk)` is equivalent to the "**`assert`**" of a property statement, and the "**`property`**" is the task. Using the **`fork / join_none`**, a new thread is initiated at every clocking event, and that thread is independent from previously launched threads.

```
always  @(posedge clk)  begin // emulate the firing of assertions
  fork
    t_ab_then_c();
    // .. t_XXX(); // firing of other emulated properties
  join_none
end
```

The task emulates the antecedent with an "**`if`**" statement to emulate the action in the attempt phase, and a "**`return`**" statement to emulate an assertion vacuity if the "**`if`**" statement is false . If the "**`if`**" statement is true, then the task proceeds to emulate the consequent. If the consequent succeed, the task completes. If it fails at anytime during the sequence of the consequent a "**`return`**" forces the exit of the task. Thus, the firing of the assertion starts a concurrent task thread at every posedge of the clock. The structure of the task is as follows:

```
task some_name();
   if(first_term_of_antecedent)  begin : attempt succeeds
      if(end_of_antecedent_sequence) begin : antecedent match
          // Test consequent
          if(end_consequent_sequence) begin :  consequent match
              // assertion succeeds
          else
             // report failure and return
            return;   // Forces an exit of the task
          end :  consequent match
      end : antecedent match
      else return; // vacuous pass, antecedent does not match
  end : attempt succeeds
```

antecedent

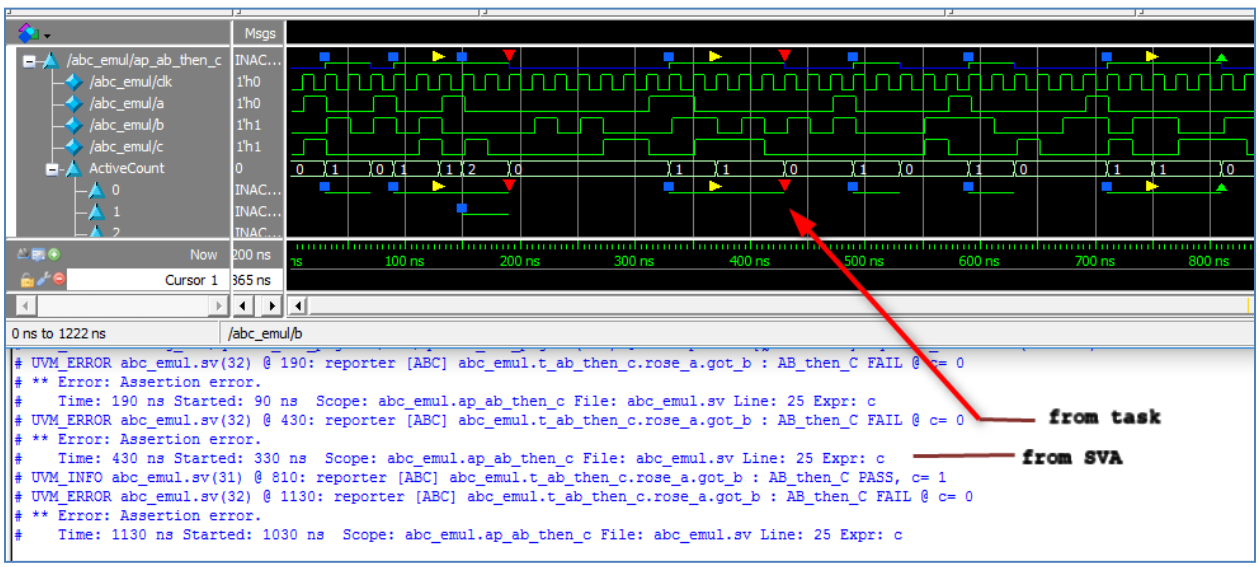
consequent

Using that structure, the property for the assertion ap_ab_then_c can be expressed as:

```
task automatic t_ab_then_c();
  if($rose(a)) begin : rose_a  // attempt succeeds
    repeat(2) @(posedge clk);
    if(b) begin : got_b        // antecedent match
       repeat(3) @(posedge clk);
       if(c) // consequent match
             `uvm_info (tID,$sformatf("%m : AB_then_C PASS, c= %b", c), UVM_HIGH)
       else  // consequent does not match
             `uvm_error(tID,$sformatf("%m : AB_then_C FAIL @ c= %b", c))
    end : got_b
    else return; // vacuous pass, antecedent does not match
  end : rose_a
endtask
```

(file http://SystemVerilog.us/vf/abc_emul.sv)

Simulation of the assertion with SVA and with the task emulation (file *abc_emul.sv*) produced the following results:



2. Emulating a complex assertion that is difficult with SVA:

Consider the same assertion as above, but with a small variation:

```
int dly1=2, dly2=7;
ap_ab_then_min_max_c : assert property($rose(a) ##dly1 b |->  ##dly2 c); // ILLEGAL SVA
```

What is desired in this assertion is having the delays be defined by dynamic values set in variables, rather than constants set at elaboration time. SVA 1800'2012 does not allow delays or repeat operators to be defined dynamically, they must be static after elaboration. A solution in SVA is to use local variables, setting them up at the successful attempt, and then use those variables as counters. Below is a possible solution, which looks rather complex.

```
// ap_ab_then_min_max_c : assert property($rose(a) ##dly1 b |->  ##dly2 c); // ILLEGAL SVA
       property p_ab_then_min_max_c;
         int v_dly1, v_dly2;
         ($rose(a), v_dly1=dly1, v_dly2=dly2) ##0
          first_match((1, v_dly1=v_dly1-1'b1)[*1:$] ##0 v_dly1 < 0) ##0 b |->
          first_match((1, v_dly2=v_dly2-1'b1)[*1:$] ##0 v_dly2 < 0) ##0 c;
       endproperty
    ap_ab_then_min_max_c: assert property( p_ab_then_min_max_c); //
```

The code is much simpler represented using tasks, s shown below:

```
task automatic t_ab_then_dly1_dly2_c();
  automatic int v_dly1, v_dly2;
  if($rose(a)) begin : rose_a
     v_dly1=dly1; v_dly2=dly2;
     repeat(v_dly1) @(posedge clk); // NO countdown and test needed
       if(b) begin : got_b
         repeat(v_dly2) @(posedge clk);
         if(c) `uvm_info (tID,$sformatf("%m : AB_then_dly1_dly2_C PASS, c= %b", c), UVM_HIGH)
         else  `uvm_error(tID,$sformatf("%m : AB_then_dly1_dly2_C FAIL @ c= %b", c))
       end : got_b
       else return;                  // vacuous assertion, exit task
    end : rose_a
    // could add "return;" // but is not needed here
    endtask

always  @(posedge clk)  begin // emulate the assertion firing
  fork
    t_ab_then_c();
    t_ab_then_dly1_dly2_c();
  join_none
end
```

2. Emulating a complex assertion that is very difficult with SVA:

The problematic issue in SVA is the flow through of local variables when dealing with ORed sequences. Another issue is how local variables are handled when they are assigned and read in multiple ORed threads. Consider these requirements that can from an actual example in the VerificationAcademy forum:

Requirements:

The requirements for this model , as demonstrated in the figure below, are:

1. Upon a rose of **go**, **data** is sent on a data bus.
2. **data** is only valid when the **vld** signal is true
3. The checksum **chksum** is on the data bus and is asserted upon the fall of **go**.
4. Following the initialization of the checker **sum** , a running sum with overflow is computed at every cycle **vld** is true.
5. In the **chksum** cycle, the checker **sum** must be compared against the **chksum** that appeared on the data bus.

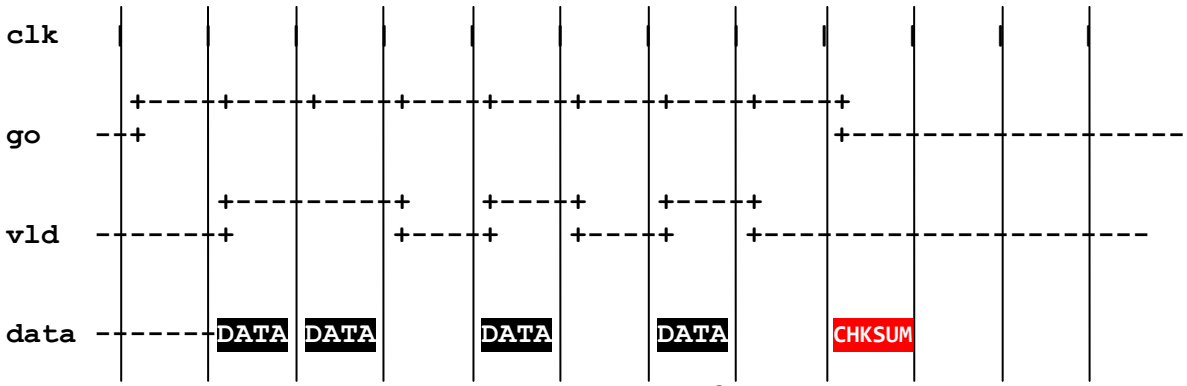


Figure 1: Requirements for assertion

Assertions that appear OK but are not!

An assertion that appears on the surface like it should work, but fails to compile, is shown below:

```
bit clk, a, b, c, go, vld;
typedef bit[3:0] BITS_4;
BITS_4 data;
event e, k; // debug
function automatic bit check_sum(BITS_4 sum, data);
  assert(sum==data)
  $display("@t=%t, sumcheck PASS, sum=%H, expected %H", $time, data, sum);
  else `uvm_error(tID,$sformatf("%m : SVA sumcheck error, sum=%H, expected %H", data, sum))
  return (sum==data);
endfunction

property p_check_msg_BAD;
  BITS_4 sum;
  bit result;
  ($rose(go), sum=0) |->
    first_match(((vld[->1], sum=sum+data)[*100000]) or
    (($fell(go)[->1], result=check_sum(sum, data)))) ##0 result;
 endproperty
 ap_check_msg_BAD: assert property(p_check_msg_BAD);
```

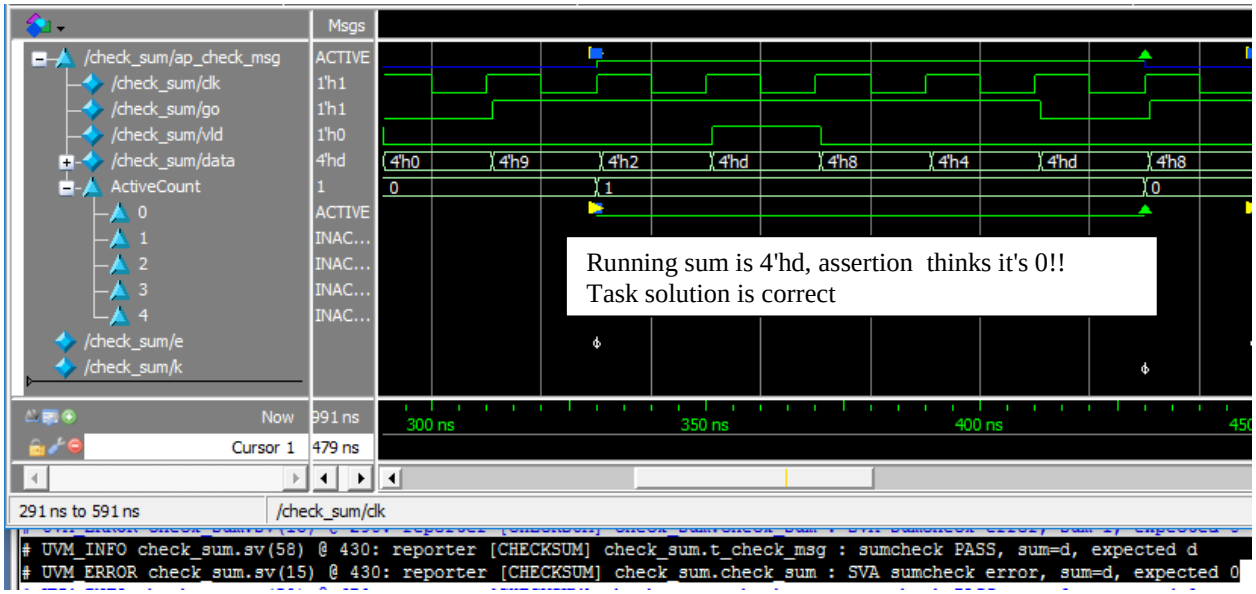
[*$] is invalid.
[*1:$] cause a match on the first occurrence of vld.
Used a repeat of large number

Local variable *result* is referenced in an expression where it does not flow.

The above assertion attempt to express that upon a rose of **go**, **sum=0**. Then, at every clock cycle **vld==1**, we compute the **sum** (local variable **sum = sum+data**). This process is repeated up to 100,000 times unless we get a fell of go, at which time we compare **sum** with the provided checksum from the **data** line. To cause an error into this property, the *chek_sum* function assigns a 0 to local variable *result* if there is a mismatch between the computed checksum and the provided checksum. However, this local variable *result* does not flow out of the ORing of the 2 sequences in the consequent.

A solution that compiles, but looks like it should work, (elaborates OK, but does not simulate correctly) is shown below. It is basically the same assertion as above, but the fail detection is in the function, thus avoiding this flow-through issue.

```
property p_check_msg; // COMPILES AND SIMULATES BUT IS IN ERROR!!!
        BITS_4 sum;
        bit result;
        ($rose(go), sum=0) |->
                first_match(((vld[->1], sum=sum+data)[*100000]) or
                        (($fell(go)[->1], result=check_sum(sum, data))));
endproperty
ap_check_msg: assert property(p_check_msg);
```



Simulation of above model (check_sum.sv).

This assertion fails because of the way local variable are used when assigned on parallel threads, and because of the flow of local variables. Specifically, (*from my SVA HAndbook)*:

Variables assigned on parallel "or" threads

🕮 **Rule:** When two sequences are ORed, each sequence produces its own thread that can get carried through to the next step, such as an end point (no more continuity), concatenation with another sequence, or as antecedent to a consequent. For example,

`(sequence1 or sequence2) ##1 seq3` *// is equivalent to*

`(sequence1 ##1 sequence3) or (sequence1 ## sequence3)` *// two threads*

Note that multithreaded sequence do occur with range operators. For example:

(a ##[1:2] b) // is equivalent to

`(a ##1 b) or (a ##2 b).`

`Also, (a ##1b[*1:2])` // is equivalent to:

`(a ##1 b[*1]) or (a ##1 b[*2]).`

If the sequence makes assignments to local variables, then <u>each of the sequence involved in the ORing carries its own individual and separate copy of the local variables</u>. Those separate copies of the local variables are carried all the way through each of the ORed sequence thread.

Thus, in this assertion example, there are two separate copies of the local variable *sum*, one during the summations when **vld** is true, and another copy for the final check, and that second copy is unaware of the first copy, Nice!

Emulating the SVA assertion with tasks

Using tasks eliminates the issues of visibility of local variables by threads, and makes the code far more readable and understandable. The following code demonstrates the concepts:

```
always  @(posedge clk)  begin
    fork
        t_check_msg();
    join_none
 end
```

```
task automatic t_check_msg();
  automatic BITS_4 sum;
  if($rose(go)) begin
    sum=0;
    ->e;    // debug
    while(go==1'b1) begin
       if(vld==1'b1) sum=sum+data;
       @(posedge clk);
    end
    // Here, go==1'b0
    // @ fell(go), data==sumcheck by transmitter
    -> k; // debug
    assert(sum==data)
    `uvm_info (tID,$sformatf("%m : sumcheck PASS, sum=%H, expected %H", data, sum), UVM_HIGH)
    else `uvm_error(tID,$sformatf("%m : sumcheck error, sum=%H, expected %H", data, sum))
    return;  // exit
  end
endtask
```

Conclusions

Concurrent SVA assertions make use of threads. Understanding the engine of SVA with tasks makes the user of assertions more sensitive to how threads are created. To implement some requirements that use local variables in ORed threads, SVA may present serious issues. The use of tasks helps resolve those issues, and in some cases, simplify the definition of assertions. Keep in mind that an assertion is just a statement that a property is true, regardless of implementation. Thus, implementing an assertion with tasks is acceptable. SVA is just a shorter notation that adapts well for most cases, but not all cases.

Referenced files:
http://SystemVerilog.us/vf/abc_emul.sv
http://SystemVerilog.us/vf/ab_then_c.png
http://SystemVerilog.us/vf/check_sum.sv
http://SystemVerilog.us/vf/check_sum.png

---

[i] Ben Cohen
http://www.systemverilog.us/ ben@systemverilog.us
For training, consulting, services: contact http://cvcblr.com/home
* [url=http://goo.gl/JOfuEB]SVA Handbook 4th Edition, 2016 ISBN 978-1518681448[/url]
* A Pragmatic Approach to VMM Adoption 2006 ISBN 0-9705394-9-5
* Using PSL/SUGAR for Formal and Dynamic Verification 2nd Edition, 2004, ISBN 0-9705394-6-0
* [url=https://goo.gl/d10QHh]Real Chip Design and Verification Using Verilog and VHDL, 2002 isbn 978-1539769712 [/url]
* Component Design by Example ", 2001 ISBN 0-9705394-0-1
* VHDL Coding Styles and Methodologies, 2nd Edition, 1999 ISBN 0-7923-8474-1
* VHDL Answers to Frequently Asked Questions, 2nd Edition ISBN 0-7923-8115