1 SUPPORT LOGIC AND THE ALWAYS PROPERTY

1.1 THE SUPPORT LOGIC

SVA has several constructs to specify requirements for concurrent events. However, there are classes of requirements where the strict use of only SVA does not support. This is because of the nature SVA modeling and also because the verification environment needs additional constructs and modeling structures to accurately define the needed properties. This support logic may include module variables of various types, such as bit, logic, associative array, queues; SVA sequences for the definition of endpoints; logic, such as assignments to these module variables; and functions called from within the properties. Application examples where support logic is needed include: 1) the uniqueness of attempts where each successful consequent does not satisfy all pending threads; 2) the existence of past occurrences of variable expressions within ranges (*finite or infinite ranges*); 3) the storage of past variable expressions. This paper addresses, by example, these three concepts.

1.1.1 Attempt Uniqueness

Every request has its own grant. This requirement assures that each successful attempted assertion from start to completion is unique; this means that if multiple assertions are active waiting for a matched consequent then a successful consequent should not satisfy all those active assertions. Consider this example:

```
ap_req_ack: assert property(@ (posedge clk)  $rose(req) |->  ##[1:10] ack );
```

If there is a **`$rose(req)`** at t1, t3, and t5 and this is followed by an ack at t7, then the t1, t3, and t5 initiated threads are satisfied by the t7 ack. To accomplish uniqueness what is desired is that each req has its own ack; thus, if there are three separate matched antecedents then we need three separate matched consequents.

To accomplish this, one could use concepts of a familiar model seen in hardware stores in the paint department. There, the store provides a spool of tickets, each with a number. As a customer comes in, he takes a ticket. The clerk serving the customers has a sign that reads "NOW SERVING, TICKET #X". The customer who has the matches ticket gets served, the others have to wait. When done, the number X in incremented, and the next in-line customer gets served. The support logic needed for this type of modeling includes:

1) Spool of tickets, modeled with the module variable `int ticket`.
2) A function `inc_ticket()` to increment the ticket. That function is called from within the property in a sequence_matched_item.
3) A "now-serving" counter, modeled with the module variable `int now_serving`.

Upon a matched antecedent, a property local variable saves the current ticket, and then increments the spool. The property waits for a matched current ticket with now-serving display. Upon termination of the assertion the `now_serving` variable is incremented in the pass or fail action block depending upon the outcome. Below is the modeling code, full testbench at this reference[i].


```
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;
```

1.1.2 past occurrences of variable expressions

A grant must at some time have been preceded by a request. This requirement addresses the case where there are erroneously acknowledges without requests. To satisfy this requirement support logic is needed to store the occurrence of a request; that occurrence is then reset with the occurrence of a grant[ii].

```
   bit clk, req, grnt, req_occurred, reset_n;
     always @(posedge clk) begin
        if(req)req_occurred <= 1'b1;
        if(grnt) req_occurred <= 1'b0;
    end

   ap_grant: assert property(@(posedge clk) $rose(grnt) |-> req_occurred);
   am_no_rerq_grnt: assume property(not(req && grnt);
```

1.1.3 storage of past variable expressions

1.1.3.1 A grant must at some time have been preceded by a requestwithin the last 5 cyles. This is check for an activity that occurred in the past 15 cycles. SVA provides a `$past` function defined as

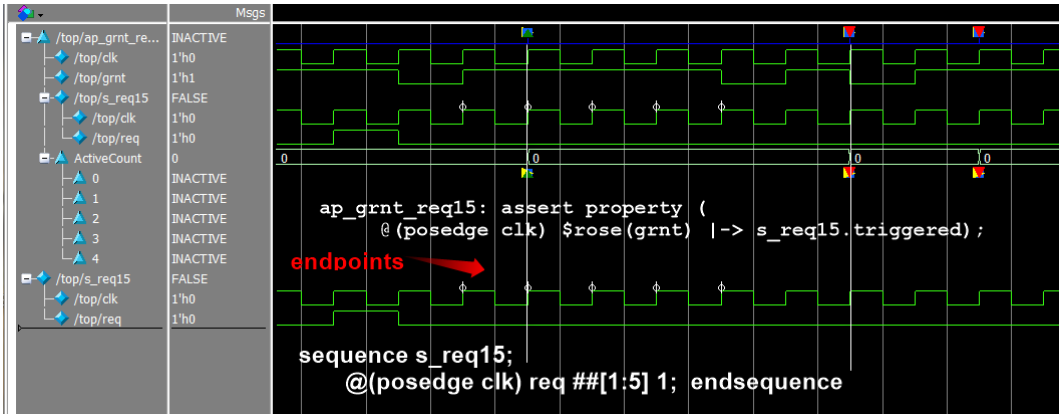
`$past``(expr1 [, number_of_ticks] [, expr2] [,clk_evnt])` where number_of_ticks is an integer and is not a range. Thus, the following is illegal:

```
$rose(grnt) |-> $past(req, 1:5); // illegal
```

To address this past range of activities we could use the endpoints of a sequence.

```
sequence s_req15;  @(posedge clk) req ##[1:5] 1;  endsequence
ap_grnt_req15: assert property (
   @(posedge clk) $rose(grnt) |-> s_req15.triggered);
```

From the simulation below one can clearly see that in the cycle where **`$rose`**`(grnt)` occurred the endpoint of `(req ##2 1)` occurred. The `s_req15.`**`triggered`** creates five endpoints for whenever `req` is true; these endpoints include `req ##1, req##2, .. req##5.`



1.1.3.2 IF a then b's signature==value

When signal A is raised at time t, it implies that signal B is 0 in the previous 3 cyles (i.e., 0 @ t-1, t-2, t-3), and it was 1 in the four cycles before that (i.e., 1 @ t-4, t-5, t-6, t-7). The simplest solution is to provide as support logic a shift register to store b's values, and use that register in the assertion. Thus,

```
bit[0:7] b_occurred;
always_ff @(posedge clk)  b_occurred <= {b, b_occurred[1:6]};

ap_ab0: assert property(@(posedge clk)  $rose(a) |->
   b_occurred[1:3] == 3'b000 && b_occurred[4:7]==4'b1111);
   // Could have wrote it as
   // b_occurred[1:7]==7'0001111
```

https://verificationacademy.com/forums/systemverilog/signals-checking#reply-107769

1.1.3.3 No two or more request without a grant

The grant occurs within a range of 1 to 5 after the request. However, this model needs to insure that there are no two or more requests without receiving a grant. Since requests occur before a grant and since each attempt is separate from other attempts, this requirement can be satisfied with the help of a support counter that counts up upon each request, and counts down upon each grant[iii]. For example:

```
    int req_count;
    always @(posedge clk) begin
        if(req)  req_count <= req_count + 1;
        if(ack)  req_count <= req_count - 1;
    end

    sequence s_req; @(posedge clk)  req ##[1:5] 1'b1; endsequence
// endpoints at 1, ..5 cycles after req
// At new ack, then req count>0 and
// there was a req in the previous 5 cycles.
    ap_ack2req: assert property(@(posedge clk)
             $rose(ack) |-> req_count > 0 && s_req.triggered);
```

1.1.4 The always

This property expression is not often used because there is an implicit *always* associated with concurrent assertions, thus allowing the assertion to be retested at each occurrence of its clocking event. However, there is a need to specify a condition under which a property always holds, or a property that must always hold within a range of cycles after an attempt. This is accomplished with the **`always`** construct. In this example, a user had the following requirements:

1) The signal `done` rises once in simulation.
2) Following `done`, signal `a` eventually goes high for one cycle and then then remains low until the end of simulation.

The user wrote the assertion in this style, which does not satisfy the requirements

```
ap_BAD: assert property (@(posedge clk) $rose(done) |=> !a[->1] ##1 !a[*1:$]);
```

That assertion is incorrect because the sequence (`!a[*1:$]`) does not check that `a` remains at zero for all remaining cycles. The repetition construct does not require that the variable remains at zero for ALL cycles, but instead it requires that it remains at zero for one or more cycles. This is a common mistake probably caused by the common use of the repeat range operator that is followed by another variable; something like:

`(b[*1:$] ##1 c)` that reads something like (`b` repeats **`until`** `c`)

`!a[*1:$]` is same as

`!a[*1[ or !a[*2] or …!a[*$];` thus any one of these sequences satisfy the end point.

A preferred solution is to use the **`always`** along with an **`initial`** construct.

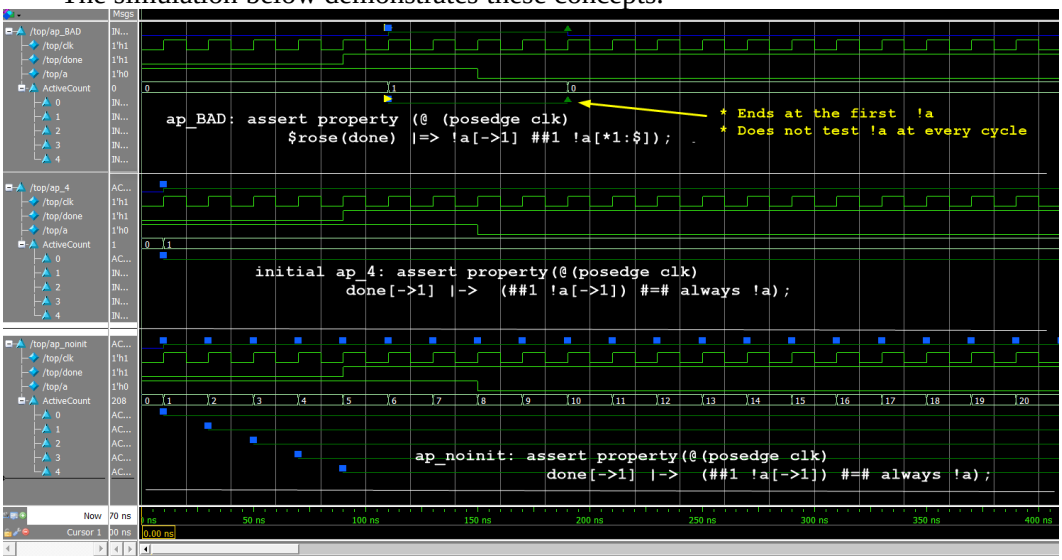
```
  initial ap_done_a_never_a: assert property(@(posedge clk)
            done[->1] |->  (##1 !a[->1]) #=# always !a);
```

Notes:

1) Because of the **`initial`** statement, there is only ONE attempt at this assertion.
2) The use of the goto operator need in this case because the antecedent is waiting for the occurrence of the **`done`** variable and there is only one attempt. In general, the goto operator as the first element of an antecedent is not recommend on concurrent assertion because they create multiple unnecessary threads; this is an exception.
3) The consequent includes a goto to await the occurrence of the variable `a`. This is then followed by the property **`always`** `!a` that verifies that `a` holds at zero for all next consecutive cycles.
4) Using the **`always`** without the **`initial`** creates multiple unnecessary threads at every attempt; thus, the user needs to be cautious when using the always property statement.

```
ap_noinit: assert property(@(posedge clk)
                done[->1] |->  (##1 !a[->1]) #=# always !a);
```

The simulation below demonstrates these concepts.



References:

1) 1800’2017: IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language
2) SystemVerilog Assertions Handbook, 4th edition … for Dynamic and Formal Verification Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari. ...and Lisa Piper
3) https://verificationacademy.com/forums/systemverilog

[i] https://verificationacademy.com/forums/systemverilog/assertion-req-and-gnt-signals

[ii] https://verificationacademy.com/forums/systemverilog/grant-must-some-time-have-been-preceded-request#reply-104623

[iii] https://verificationacademy.com/forums/systemverilog/assertion-protocol-checker