Understanding the SVA Engine Using the Fork-Join Model

SVA (*SystemVerilog Assertions*) is a powerful short-handed assertion language with many constructs; it is built as an integral part of SystemVerilog but with a specific syntax and sets of rules. Unlike a scoreboard that tends to focus on a model implementation that mimics the DUT, SVA addresses the requirements; that brings out a better understanding of the requirements, along with its weaknesses for lack of definitions.

SystemVerilog

Ben Cohen Last Updated Jul 2020

Antecedent, Assertions, Consequent, Fork-Join, Properties, SystemVerilog, July 2020 - Volume 16 Issue 2, Verification Horizons, Beginner

Mark as completed

INTRODUCTION

SVA (*SystemVerilog Assertions*) is a powerful short-handed assertion language with many constructs; it is built as an integral part of SystemVerilog but with a specific syntax and sets of rules. Unlike a scoreboard that tends to focus on a model implementation that mimics the DUT, SVA addresses the requirements; that brings out a better understanding of the requirements, along with its weaknesses for lack of definitions. Over the years, I experienced the difficulties that engineers have in the thorough understanding of the SVA underlying model and why sometimes the assertions behave unexpectedly to the users' intent.

This article is a follow-up to the *SVA Alternative for Complex Assertions* article published in the March 2018 issue of *Verification Horizons*. Unlike that issue that stressed the use of tasks to model complex assertions, this article explains SVA through the modeling of the underlying principles of some of its core elements using SystemVerilog procedural constructs. This modeling style emphasizes the concepts of "threads", as typically demonstrated in debug tools. The focus is in the modeling of threads for properties where both the antecedent and the consequent are sequences with range delays. Note that the simulation implementations use optimization features, thus differing from what is presented in this article; however, the underlying principles presented here are still valid.

Three different types of SVA properties are used to emphasize different important concepts of SVA:

1. **Antecedent/consequent:** This model demonstrates the concepts of spawned threads, leading clocking event, attempts.
2. **Range delays in the consequent:** This model demonstrates the testing of each element of the range until success is reached and the lack of need to limit the ranges to a matched (i.e., valid) consequent.
3. **Range delays in the antecedent:** This model demonstrates the testing of each element of the range and the need for a ***first_match*** operator or other technique

to limit the ranges to a matched antecedent.

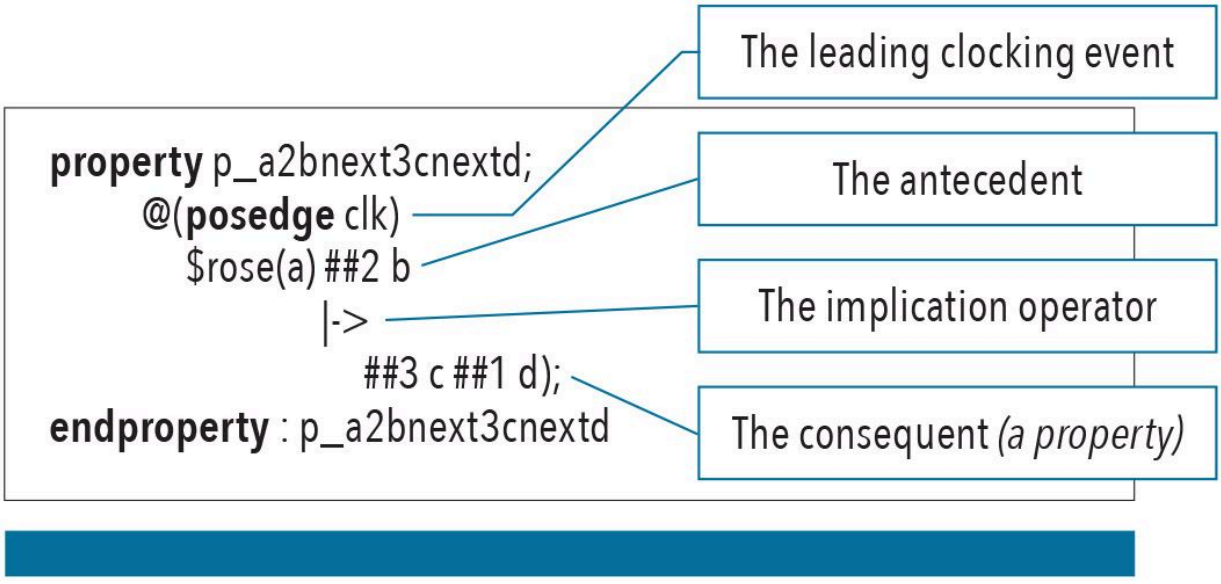
This inner depth of SVA modeling provides a greater understanding of SVA and that appreciation will make you a more efficient SVA coder. It also makes you appreciate the value of SVA short-hand notation and how SVA is more concise and readable when defining design properties. As a bonus, this assertion modeling approach provides more insights when used in the verification process within SystemVerilog classes or to solve complex verification problems that are difficult to implement in SVA.

Note that this modeling approach maintains the spirit (*but not vendor's implementations*) of SVA. Also note that it is possible to use the task modeling approach with fewer threads, such as using the **while** loop instead of launching threads.

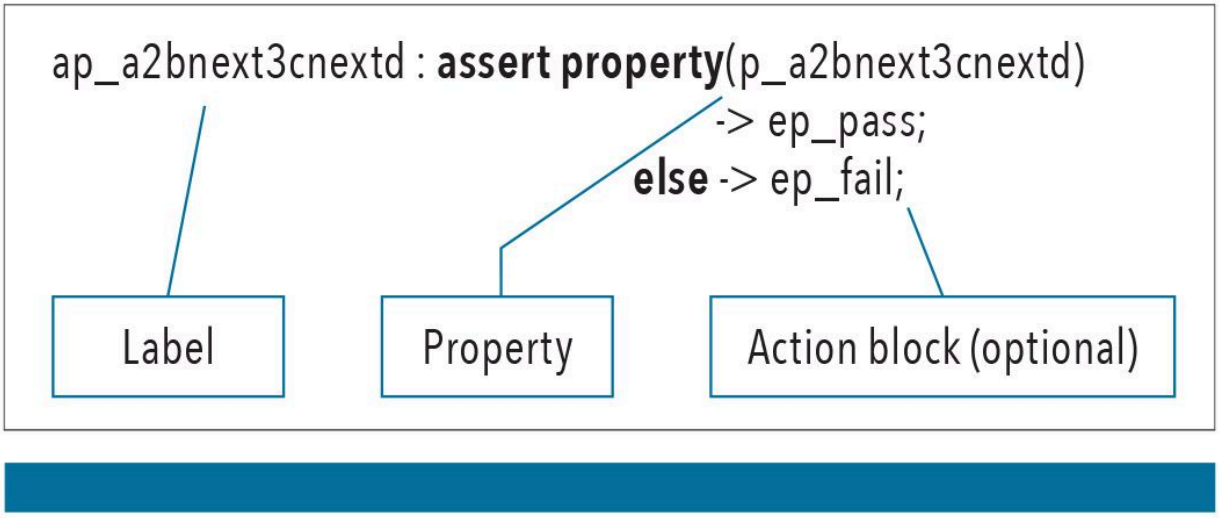
FUNDAMENTAL CONCEPTS OF A PROPERTY AND AN ASSERTION

Before getting into the modeling aspects two definitions are needed:

Property: A property is a characteristic of the design or the body of the requirements; it is a collection of logical and temporal relationships between and among subordinates. Thus, a property is a statement about a design's intended behavior; it does not simulate. For example:



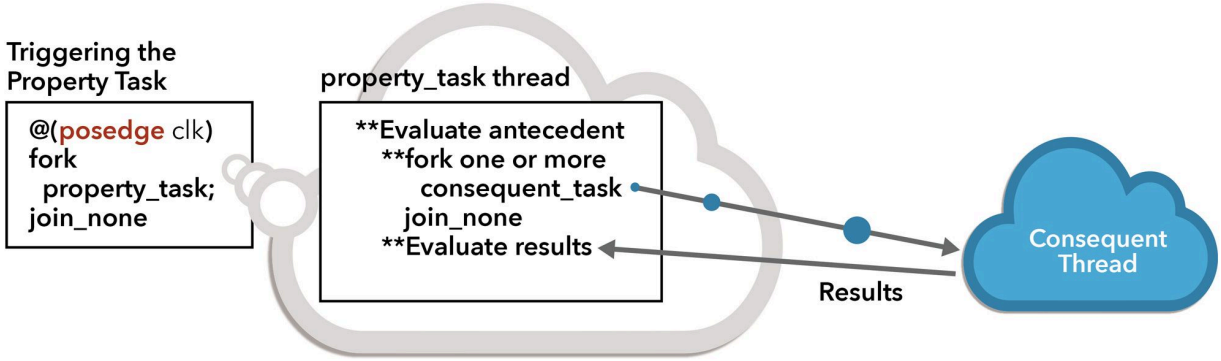
Assertion: A statement or directive that a given property is required to hold (be true) during its execution cycles. Thus, an assertion is a simulation/verification statement to execute the concurrent property as a parallel block.



Since a property represents a block statement that executes as a parallel block when triggered it can be emulated using tasks that are spawned in **fork-join_none** blocks. SystemVerilog *1800'2017:9.3.2 Parallel blocks* states that the **fork-join** parallel block construct enables the creation of concurrent processes from each of its parallel statements. If the task is **automatic**, it is dynamically created during the forking, it executes concurrently with other blocks, and it dies when completed. In many respects it parallels an SVA evaluation in that the assertion is also dynamically created in the attempt phase (i.e., the start), it runs as a concurrent process, and it dies when completed.

Thus, a spawned **automatic** task creates a dynamic thread that contains a set of possible sequential statements and can spawn other secondary threads with other **fork/join_none** tasks. Each of those dynamically created processes has a life of its own, independent from other threads. The figure below demonstrates the high-level model for the assertion with an antecedent, an implication operator, and a sequence for the consequent.

Figure 1 - High-level model for an assertion



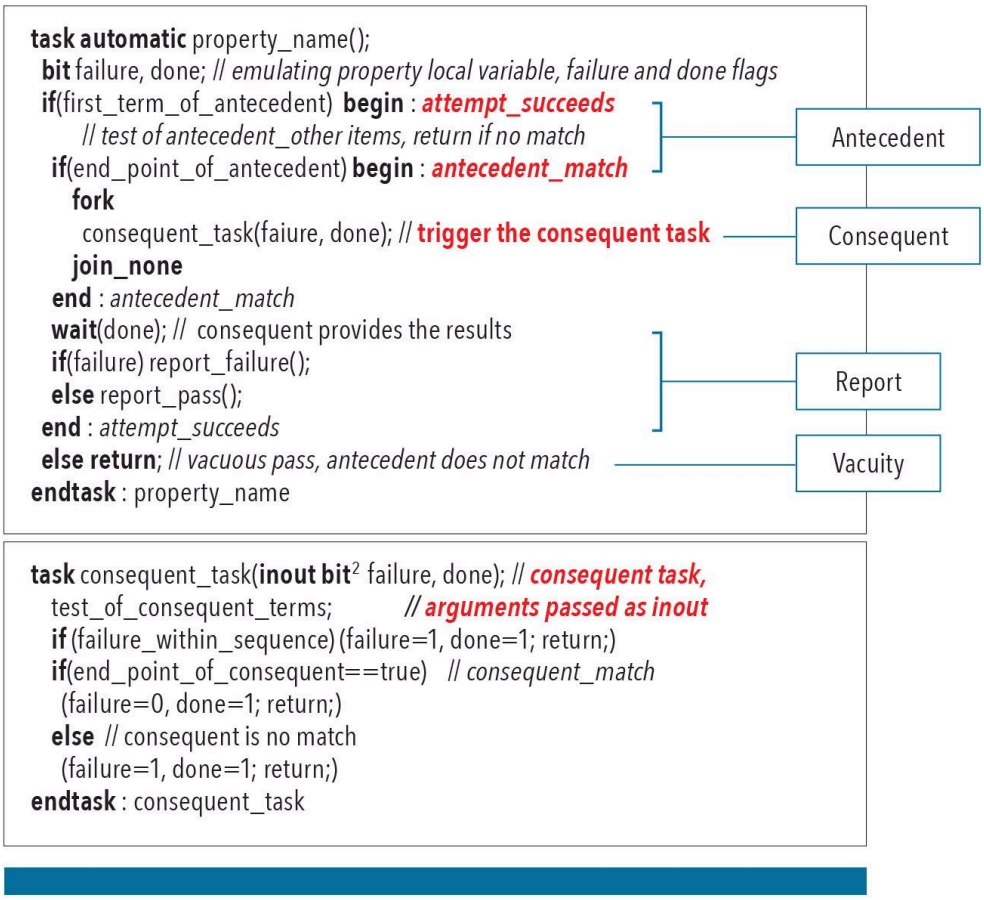
1. **Property triggering:** At every clocking event, the **automatic** task that represents the model for the property is forked.
2. **Task evaluation:** The code representing the antecedent is evaluated. It the antecedent is nonvacuous then one or more consequent threads are forked. In SVA, properties that include range delays and sequence-repeats are multi-threaded; that would be modeled with tasks, one task for each thread. Those separate forked tasks do represent a better representation of the SVA model, though writing those models in that fashion can get very complex. Depending upon the type of property, it is possible to write the code with fewer tasks, thus varying a bit from the "true" SVA model, but still be correct. *This article will use that simpler code approach wherever it is appropriate.*
3. **Result evaluation:** Three possible results are possible for an assertion: Vacuous, Pass, or Fail. Vacuity is determined in the evaluation of the antecedent; a vacuous result is considered true (*vacuously*) as opposed to a truly valid pass where the antecedent and consequent are both matches. Pass or Fail of the property can in some instances be determined during the evaluation of the consequent. However, in many multithreaded cases the result from each consequent thread must be collected to determine the final assessment of the assertion. Thus, in this model, each consequent thread sends its results to the master thread (i.e., the antecedent thread).

The task SVA model

A property can be emulated with an automatic task. The first expression of the antecedent is evaluated with an **if** statement that represents the attempt phase, and a **return** statement to emulate an assertion vacuity when that **if** condition is false. If the if condition is true (*i.e., a successful attempt*), then the task proceeds to emulate further sequence elements until it reaches the antecedent endpoint, but checking at each clocking event that the sequence still matches. If at any cycle in the sequence, there is no match then a **return** statement emulates vacuity.

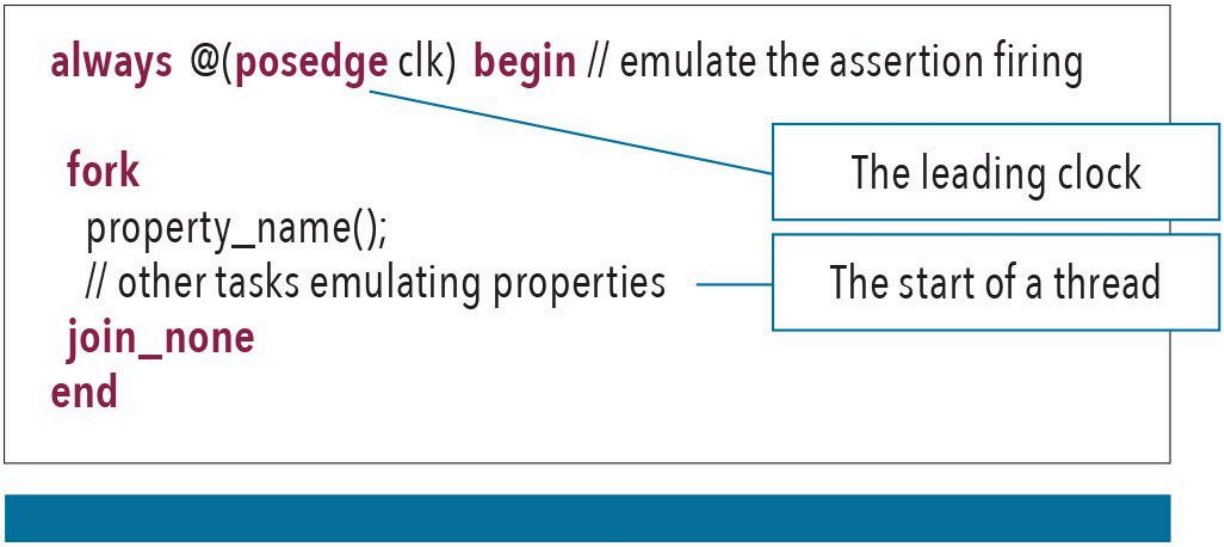
At the nonvacuous successful completion of an antecedent thread (*i.e., reaching an end point of a sequence*) the task proceeds to emulate the consequent. Since the consequent is a property, it is also emulated using another task that is spawned using a **fork/join_none**. If an antecedent is multithreaded with a range or a repeat construct, each thread of that antecedent may trigger a separate consequent thread; this is demonstrated further down in this article. The consequent thread reports its success or failure to the calling task so that it may decide to short-circuit the evaluation in case of failure, or to consider all the evaluations before considering a success. The generic structure of the task with pseudo-code is shown in the two code samples below.

Generic structure emulating a SVA property



The assert model

// Emulation of the assert statement



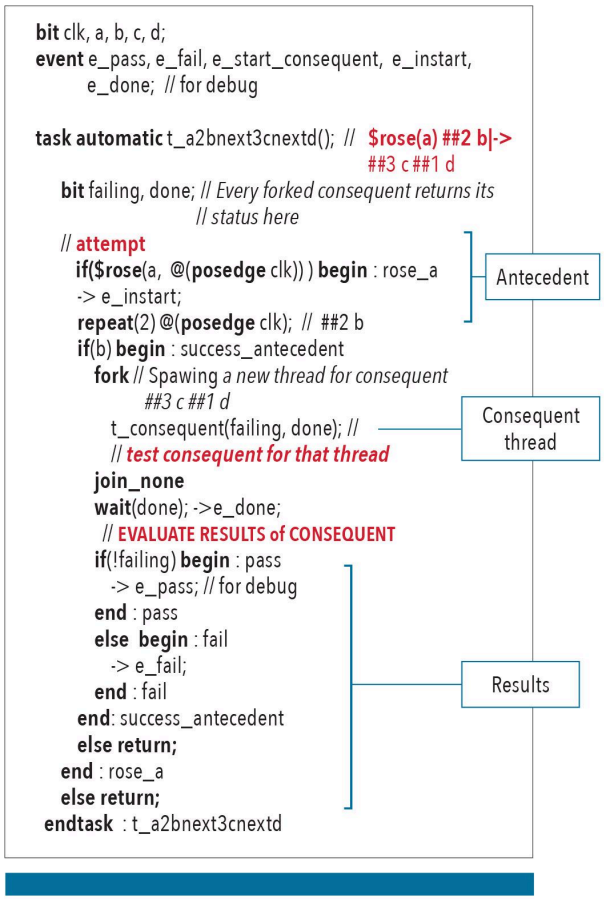
In the model of the **assert** statement shown above the **task** property_name() is triggered in parallel to other possible tasks using the clocking event **posedge** clk.

Simple Assertion

Consider the following simple assertion:

```
ap_a2bnext3cnextd : assert property(@(posedge clk) $rose(a) ##2 b|-> ##3 c ##1 d);
```

The property of that assertion states that if the attempt of **$rose**(a) is true then **b** is evaluated two cycles later. If the end point b is true then the consequent is immediately evaluated as a separate thread. When the results of the consequent thread is received, provide the report. The task modeling is shown below.



The consequent task implements the sequence thread (##3 c ##1 d). It states that for the property, represented by a sequence, after 3 cycles the variable c must be true. This must then be followed after 1 cycle by d to also be true. The consequent task is shown below.

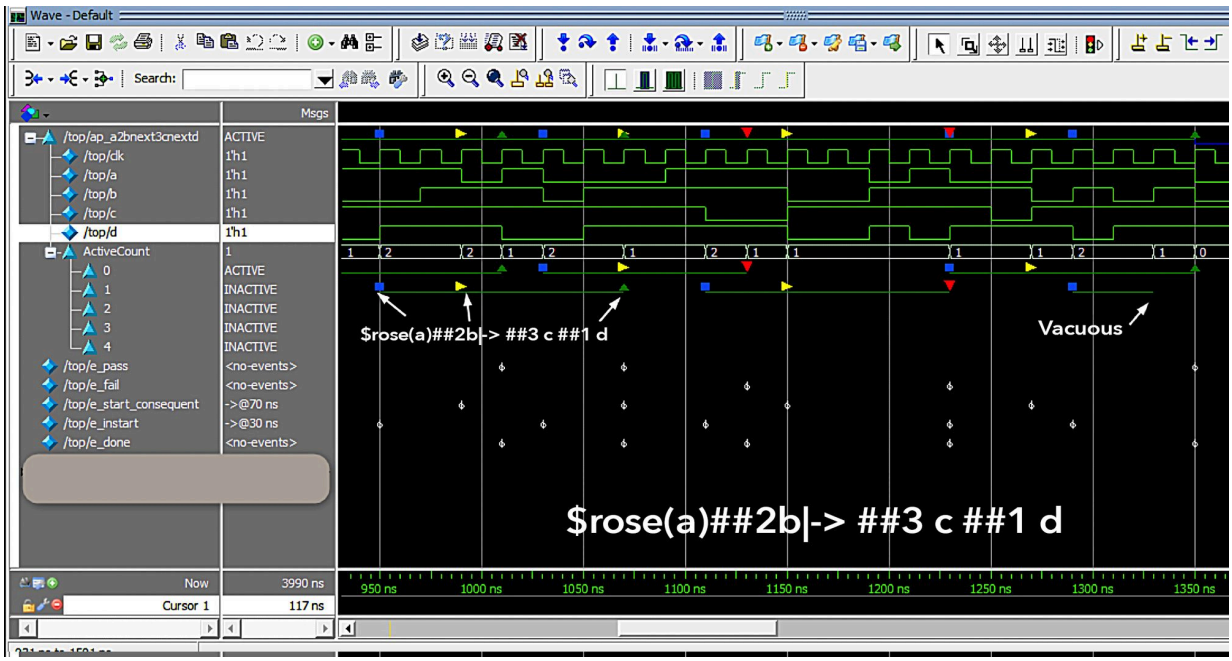
```
// $rose(a) ##2 b |->  ##3 c ##1 d);
 task automatic t_consequent (inout bit failing, done);
     repeat(3) @(posedge clk);  // ##3 c
     if(c) begin : c_pass
       repeat(1) @(posedge clk); // ##1 d
       if(d) begin : d_pass
         failing=1'b0; // flag for reporting back to calling task
         done=1'b1;  // tell calling test of done
       end : d_pass
       else begin : d_fail
         failing=1'b1; // flag for reporting back to calling task
         done=1'b1;
       end : d_fail
     end : c_pass
     else begin : c_error
       failing=1'b1; // flag for reporting back to calling task
       done=1'b1;
     end : c_error
   endtask : t_consequent
```

Using the task modeling structure defined above, the assertion can be expressed as:

The consequent task implements the sequence thread (##3 c ##1 d). It states that for the property, represented by a sequence, after 3 cycles the variable c must be true. This must then be followed after 1 cycle by d to also be true. The consequent task is shown below.

```
always @(posedge clk) begin // emulate the assertion firing
    fork
      t_a2bnext3cnextd();      // Assertion for the above
                               // property emulated with a task
      // .. other tasks representing properties
    join_none
  end
```

Figure 2 - Simulation of $rose(a) ##2 b|-> ##3 c ##1 d (ap_a2bnext3cnextd.sv, a2b3c1d_simple.png)



Variation with a range in the consequent

Consider a variation to the previous model

```
ap_a2bnext3cnextd_range : assert property(@(posedge clk)
    $rose(a) ##2 b|-> ##[1:3] c ##1 d);  // range in consequent
```

This property uses the same antecedent as the previous assertion. However, in the consequent instead of (a ##3 c ##1 d) we have a range in the c, specifically (##[1:3] c ##1 d). The ##[1:3]c ##1 d is equivalent to ((##1 c ##1 d) or (##2 c ##1 d) or (##3 c ##1 d)), and that sequence (*with the sequence ORing*) represents three separate threads launched when the antecedent succeeds.

For the assertion to succeed, the antecedent must match and any thread in the consequent must match. For an assertion failure, all three threads of the consequent must fail.

The modeling is identical to the previous model with the exception of the consequent task t_consequent that forks three separate tasks (*the threads*) with each one passing its thread results. Those results are then collected in the t_consequent task local variables. When all the forked threads are completed with the wait({done1, done2,

done3} == 3'b111) the analysis of the results is then reported back to the antecedent thread for final reporting.

Note that this modeling approach uses a style to emphasize the notion of threads and does not necessarily represent a simulation implementation.

The modeling changes include the following two tasks:

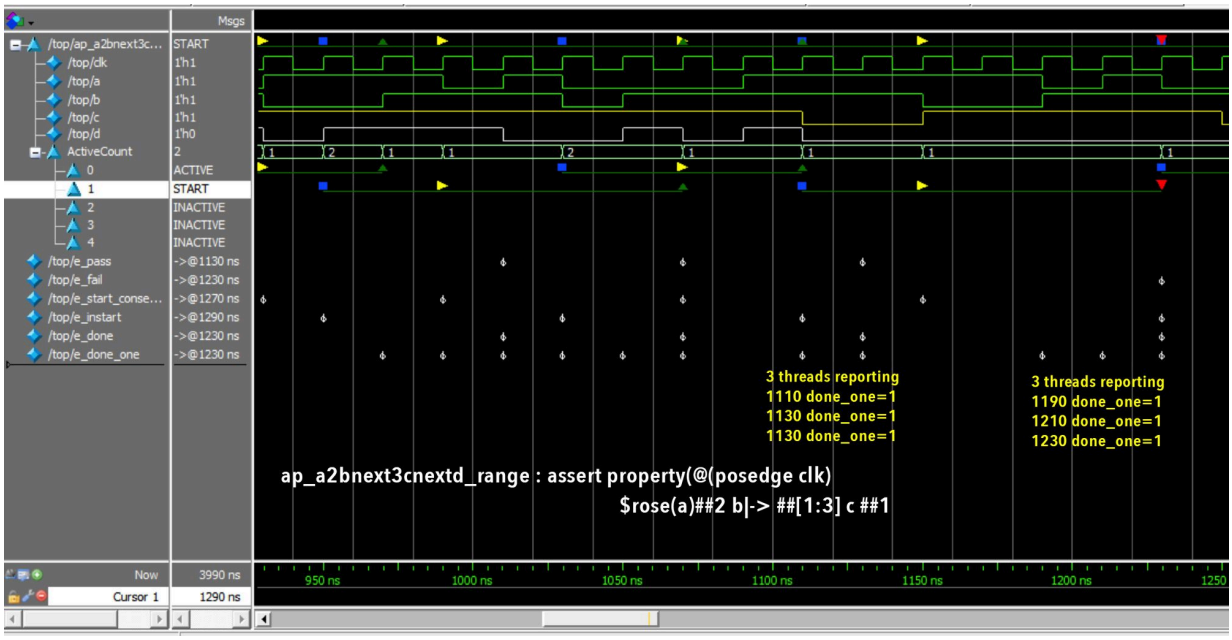
```
task automatic t_consequent (inout bit failing, done); // ##[1:3] c
                                                          ##1 d);
    bit done1, done2, done3, fail1, fail2, fail3;
    -> e_start_consequent;  // debug
    fork
      t_range(1, fail1, done1);  // (##1 c ##1 d) thread
      t_range(2, fail2, done2);  // (##2 c ##1 d) thread
      t_range(3, fail3, done3);  // (##3 c ##1 d) thread
    join_none
    wait({done1, done2, done3} == 3'b111); // all done

    if(fail1 && fail2 && fail3)  begin : a_failure //
                                        all iterations failed
      failing=1'b1; // flag for reporting back to calling task
      done=1'b1;  // tell calling test of done
    end : a_failure
    else begin : a_pass
      failing=1'b0; // flag for reporting back to calling task
      done=1'b1;
    end : a_pass
  endtask : t_consequent
```

Model for ($rose(a) ##2 b|-> ##[1:3] c ##1 d), File: ap_a2bnext3cnextd_range.sv

```
task automatic t_range(int iteration, inout bit fail, done_one);
    repeat(iteration) @(posedge clk);  // ##[iteration] c
    if(c) begin : c_pass
      repeat(1) @(posedge clk); // ##1 d
      if(d) begin : d_pass
        fail=1'b0; // PASS
        done_one=1'b1;
        -> e_done_one; // debug
      end : d_pass
      else begin : d_fail //
        fail=1'b1; //
        done_one=1'b1;
        -> e_done_one;
      end : d_fail
    end : c_pass
    else begin : c_error
      fail =1'b1; //
      done_one =1'b1;
      -> e_done_one
    end : c_error
  endtask
```

The simulation of that code is shown below in figure 3.

Figure 3 - Simulation (file ap_a2bnext3cnextd_range.png)



Demonstrated SVA concepts

1. **Leading clocking event:** SVA requires that every assertion must have a leading clocking event to start the evaluation of the property; in the examples, it is

expressed by the @(posedge clk) as the first component of the SVA assertion statement. That "start" of the assertion is called "the attempt"; in the above assertions this is the evaluation of the **$rose(a)**. Thus, one can say that at every leading clocking event there is an attempt that starts a thread of evaluations over zero or more cycles. In the task modeling, the leading clocking event is the @(posedge clk) in the block that initiates the task. The attempt is the evaluation of the first element of the property, which can contain the evaluation of other variables over one or more cycles. Since no other clocking event is specified, all clocking delays (e.g., ##2) use the same clocking event as the one used in the attempt phase.

2. **Threads:** A new thread is started to evaluate the property (*defined as a task*). This is emulated with the forked task t_a2bnext3cnextd(). Within that thread, if the antecedent succeeds, the consequent thread is initiated, emulated with t_consequent forked task. If the consequent has a range, then within that last forked thread it forks one or more new thread. This is demonstrated in the example with the forking of the following tasks: t_range(1, fail1, done1); t_range(2, fail2, done2); t_range(3, fail3, done3);

Arguments are included to identify the range number needed for each of the forked threads, and for the thread to report back its results using the **inout** direction, specifically: **task automatic** t_range**(int** iteration, **inout bit** fail, done_one**);**

Important: Why is understanding the concept of threads and thread creation important? It's because each thread represents a concurrent process that is dynamically created, and this can explain why an assertion may behave unexpectedly as a result of multiple matches even after a first match. The table below shows examples of code that solve some multithread issues.

Instead of	*Write this*
a[->1] |-> b; // same as !a[*0:$] ##1 a |-> b	a |-> b; // or better yet $rose(a) |-> b

a ##[1:$] b |-> c; // all antecedent threads
// must be tested, cannot succeed but can fail
	a ##1 b[->1] |-> c; // can succeed or fail

a ##[1:90]b ##[1:90]c |-> ##[1:$]d ##[1:50] c;	Lots and lots of threads! Can the requirement be expressed with multiple simpler assertions? Can these ranges be reduced?

Range delays in the antecedent

Consider the following property:

```
$rose(a) ##[1:5] b |-> ##3 c;
```

It states that at each end point of the sequence ($rose(a) ##[1:5] b) the variable "c" will be true three cycles later.

The issue that arises with many users is that the antecedent is multithreaded, and that can causes unanticipated errors. SVA requires that each of the threads of that antecedent with a range or a repeat statement, must be tested with its appropriate consequent.[3]

Specifically, the property:

```
$rose(a) ##[1:5] b |-> ##3 c; is equivalent to ($rose(a) ##[1] b) or ($rose(a) ##[2] b) or ..(
```

This ORing in the antecedent creates multiple threads, something like:

```
($rose(a) ##[1] b |-> ##3 c) and // separate thread
($rose(a) ##[2] b |-> ##3 c) and // separate thread
($rose(a) ##[5] b |-> ##3 c);// last thread
```

Note: Any non-matched thread of the antecedent (*when b==0*) creates a vacuously true property result for that thread, meaning that though TRUE, it is of no significance. However, any failure of one of those property threads (*Antecedent through consequent*) causes the assertion to fail. That would occur if one of the treads of the antecedent is a match and its corresponding consequent is not a match (*when c==0*). For the property to success, there can be no property failure (*vacuity is considered a vacuous true*), and at least one thread of the antecedent matches along with a match of its corresponding consequent. Below, and on the following page, is a SystemVerilog model; admittedly it is a bit complicated!

```
bit clk, a, b, c;
  event e_pass, e_fail, e_1to5end, e_in_t3, e_inloop,
        e_instart;  // for debug
  event ep_pass, ep_fail, e_done;  // for debug
  let tID="My model";
  ap_a1to5b_then_3c : assert property(@(posedge clk)
                          $rose(a) ##[1:5] b |->  ##3 c);
```

```
// The Consequent thread, ##3 c
task automatic t_3c(bit last_iteration, inout bit vfailing,
                    vdone); // ##3 c
   -> e_in_t3;
   repeat(3) @(posedge clk); // in the consequent
   if(!c) begin : thread_fail
    vfailing=1'b1; // flag the calling task of a failure
                   // for reporting
     -> e_fail;  // for debug
     vdone=1'b1; // Flag the calling task that a result
                 // was found because of failure
   end : thread_fail
   else -> e_pass;  //   thread_passes,
                    //   but other threads may exist
   if(last_iteration) vdone=1'b1;   // All threads are tested
 endtask
```

```
Model for ($rose(a) ##[1:5] b |-> ##3 c);
```

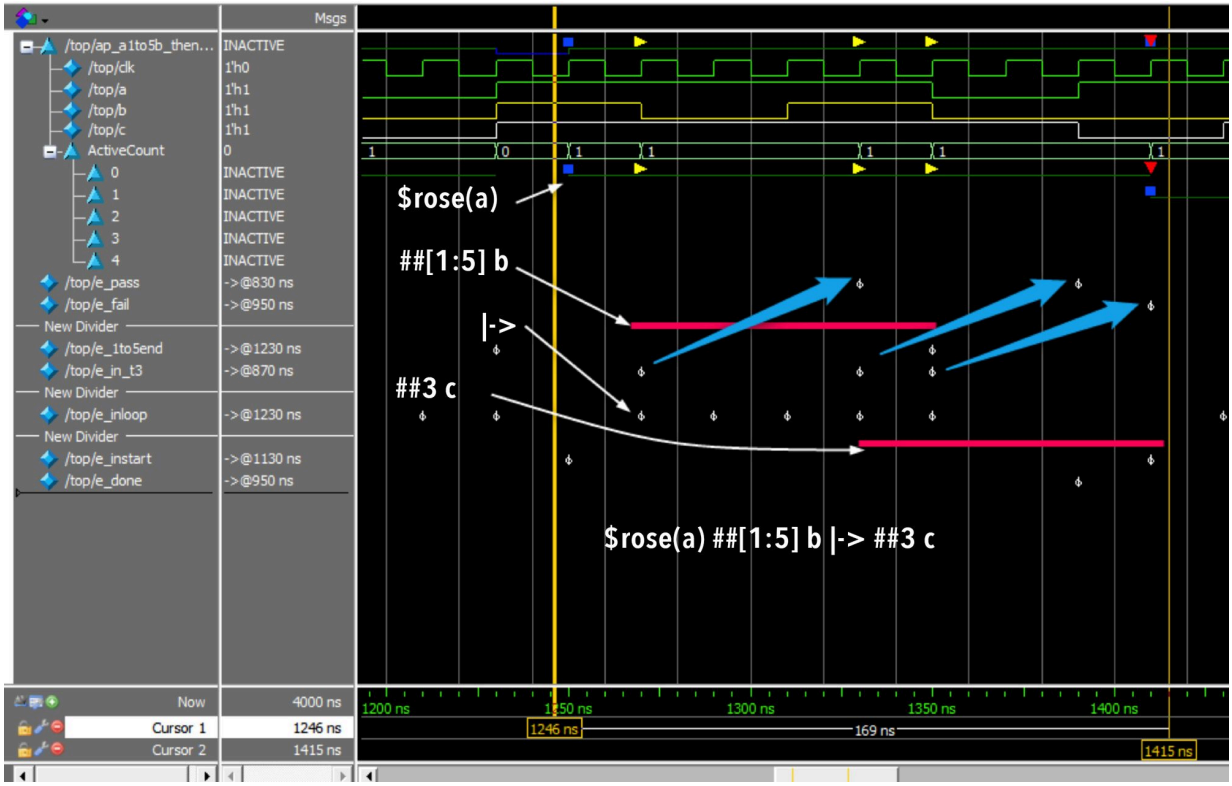
The simulation of this code above is shown in figure 4 below.

Figure 4 - Simulation results File a_range_one.png

```
// The attempt thread
task automatic t_a1to5b_then_3c();// $rose(a) ##[1:5] b |
                                  // -> consequent
 bit failing, done, last_iteration; // Every forked consequent
                                    // returns its status here
  if($rose(a, @(posedge clk)) ) begin : sig_a // attempt
   -> e_instart; // for debug
  for (int i=1; i <= 5; i++) begin : the_1to5  // Loop to cover
                                               // all threads
    last_iteration= (i==5);  //  determine the last iteration
                             //  for send to consequent
    @(posedge clk); // The ##1 for the ##[1:5]
     -> e_inloop;
    if(b) begin : got_b        // antecedent end point for
                               // that thread
    // but SVA requires all threads of antecedent to be tested
     fork // Spawning a new thread for each
          // nonvacuous element of the range
       t_3c(last_iteration, failing, done); // test consequent
                                            // for that thread
     join_none
    end : got_b
  end : the_1to5
   -> e_1to5end;
  wait(done);  -> e_done;
    // done==1 if consequent fails or the last thread of consequent
  if(!failing) begin : pass
    -> e_pass; // for de bug
  end : pass
  else  begin : fail // report the failure
    -> e_fail;
  end : fail
 end : sig_a
 else return; // vacuous assertion, ($rose(a)==0, exit task
endtask
```

Error reporting and action block

```
// The assert
 always @(posedge clk)  begin // emulate the assertion firing
   fork
     t_a1to5b_then_3c();  // Assertion for the above property
                          // emulated with a task
     // .. other tasks representing properties
   join_none
 end
```



In most cases, a user is interested in the evaluation of the first match of the antecedent with its consequent, instead of all threads of a multi-threaded antecedent. This is particularly important if the range delay is infinite (e.g., ##[1:$] because the

assertion can never succeed since ALL threads of antecedent must be tested for success and the number of threads is infinite; it can fail though. To accomplish this in SVA one can use the first_match function to exclude all other matches after the first match of the antecedent sequence. The SVA property is expressed as:

```
ap_FMa1to5b_then_3c : assert property(@(posedge clk) first_match($rose(a) ##[1:5] b) |-> ##3 c
```

The modeling of such a property is identical to the previous one with two exceptions:

1. The consequent task informs the calling task (the antecedent) that it ended with a pass or a failure. Without the **first_match** operator, it was the last thread of the antecedent (***last_iteration***) that informed its consequent thread when it was done with its execution if there were no failure in the thread.
2. The spawning of consequent threads stops (with a **break** of the loop) when a **first_match** is reached.

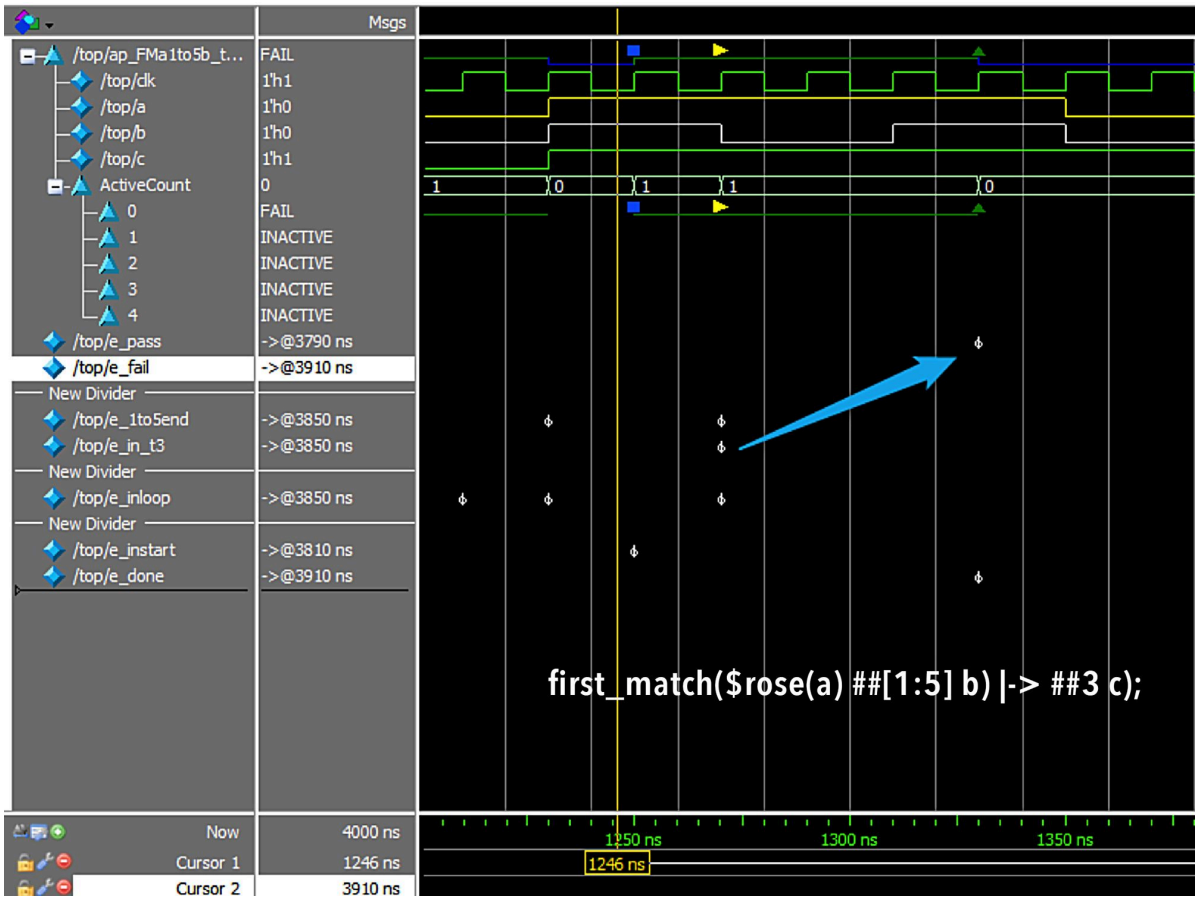
```
task automatic t_3c(bit last_iteration, ref bit vfailing, vdone);
                    // ##3 c
    repeat(3) @(posedge clk); // in the consequent
    if(!c) begin : thread_fail
     vfailing=1'b1;  vdone=1'b1;
    end : thread_fail
    else -> e_pass;  //   thread_pass
    // if(last_iteration) <<<<< DELETED for first_match()
    // because one match in antecedent was found
    vdone=1'b1;
  endtask
```

```
task automatic t_a1to5b_then_3c();// first_match($rose(a)
                                  ##[1:5] b) |-> consequent
  .... ($rose(a) the loop for ##[1:5]
      if(b) begin : got_b          // antecedent end point
       fork // Spawing a new thread for each of the element
            // of the range
         t_3c(last_iteration, failing, done); // test consequent
                                              // for that thread
       join_none
       break; //  Do not started other threads,
              //  found a first_match() in antecedent
      end : got_b
...
 endtask  //
```

```
Model for (first_match($rose(a) ##[1:5] b) |-> ##3 c);
```

The simulation for the code above is shown in figure 5.

Figure 5 - Model Simulation a_range_one.sv a_range_oneFM.png



Demonstrated concepts - Multi-threading and need for unique match in antecedents

The sequence ##[1:n]sequence creates multiple threads. This model demonstrates that for assertions with range delays in antecedents there is a need for a unique match (*refer to the_1to5* ***for*** *Loop*). This is because SVA requires that each of the threads of a multithreaded antecedent be tested with its appropriate consequent, and for success, each of those threads must succeed.

Note that this multithreaded concept is not exclusive to delay ranges, but also apply to repeat ranges.

For example:

```
ap_repeat_1to5b_then_3c : assert property(@(posedge clk) ($rose(a) ##1 b[*1:5] ) |-> ##3 c); /
```

In many cases, the first_match() function can be avoided with the use of the goto operator ([->1].

For example, instead of:

```
first_match($rose(a) ##[1:$] b) |-> ##3 c;
```

Use:

```
$rose(a) ##1 b[->1]) |-> ##3 c; // Use the goto operator
```

Equivalent to:

```
$rose(a) ##1 !b[*0:$] ##1 b)|-> ##3 c;
```

CONCLUSIONS

SVA simplifies the coding of concurrent assertions; attempting to emulating them with SystemVerilog tasks can be complex, tedious, and error prone. However, emulating the assertions with tasks provides a deeper understanding of key concepts and guidelines including the notion of leading clocking events, vacuity, spawning of threads, constraining each antecedent to a unique sequence (*instead of multiple or infinite ranges*).

The use of tasks can be used for the verification of complex assertions higher level blocks such as unit tests. Another benefit of tasks is that they can be used in SystemVerilog classes where the use of concurrent SVA is illegal.

REFERENCED FILES

1. SystemVerilog.us/vf/ap_a2bnext3cnextd.sv
2. SystemVerilog.us/vf/a2b3c1d_simple.png
3. SystemVerilog.us/vf/ap_a2bnext3cnextd_range.sv
4. SystemVerilog.us/vf/ap_a2bnext3cnextd_range.png
5. SystemVerilog.us/vf/a_range_one.sv
6. SystemVerilog.us/vf/a_range_one.png
7. SystemVerilog.us/vf/a_range_oneFM.sv
8. SystemVerilog.us/vf/a_range_oneFM.png

END NOTES

1. SVA Handbook 4th Edition, 2016 ISBN 978-1518681448. ben@SystemVerilog.us
2. SystemVerilog has the restriction that an actual argument passed as reference (with the ref) cannot be used within fork-join_any or fork_join_none blocks. See: https://verificationacademy.com/forums/systemverilog/actual-argument-passed-reference-cannot-be-used-within-fork-joinany-or-forkjoinnone

3. https://verificationacademy.com/forums/t/the-way-how-threading-works-in-assertions/37143#reply-87173v

Acknowledgement: I thank my co-author Srinivasan Venkataramanan from VerifWorks for his valuable comments and expertise.

Back to Top

July 2020 Articles

Formal is the New Normal - Deploy These FV Apps in Your ... Formal V... Article

Formal verification is now pervasive in many chip design verification projects. Key to this widespread adoption is the availability of automated “apps” that makes it easy to deploy Formal in hitherto simulation-only projects. We at VerifWorks have a long history of formal deployment at many design houses and have seen the challenges engineers face while adopting the same. We have also trained hundreds of engineers to use Formal with ABV (Assertion-Based Verification) through CVC.

Understanding the SVA Engine Using the Fork-Join Model SystemVe... Article

SVA (*SystemVerilog Assertions*) is a powerful short-handed assertion language with many constructs; it is built as an integral part of SystemVerilog but with a specific syntax and sets of rules. Unlike a scoreboard that tends to focus on a model implementation that mimics the DUT, SVA addresses the requirements; that brings out a better understanding of the requirements, along with its weaknesses for lack of definitions.

Bridging the Portability Gap for UVM SPI VIP Core Reuse From IP t... U... Article

This article focuses mostly on the vertical reuse of the test intent from IP-block to Sub-System and study of reusability from Sub-system to SoC level. The example taken to demonstrate vertical reusability is a single master and slave SPI Core IP configuration. A UVM layered testbench is wrapped around the design to verify and validate proper functioning of SPI Core IP.

PCIe® Simulation Speed-Up with PLDA PCIe® Controller for ... Verificati... Article

PCI Express® (PCIe®) is a dominant technology for hardware applications requiring high-speed connectivity between networking, storage, FPGA, and GPGPU boards to servers and desktop systems. It is a robust technology that has evolved over decades to keep up with advancements in throughput and speed for I/O connectivity for computing requirements.

Extending SoC Design Verification Methods for RISC-V Pro... Verificati... Article

As SoC developers adopt RISC-V and the design freedoms that an Open ISA (Instruction Set Architecture) offers, DV teams will need to address the new verification challenges of RISC-V based SoCs. The established SoC verifications tasks and methods are well proven, yet depend on the industry wide assumption of ‘known good processor IP’ based on the quality expectations associated with IP providers such as Arm or MIPS Technologies.

Addressing VHDL Verification Challenges with OSVVM

VHDL 2008 Article

Most people don't think of VHDL as a verification language. However, with the Open Source VHDL Verification Methodology (OSVVM) utility and verification component libraries it is. Using OSVVM we can create readable, powerful, and concise VHDL verification environments (testbenches) whose capabilities are similar to other verification languages, such as SystemVerilog and UVM.

Effective Validation Method of Safety Mechanism Complia...

Function... Article

The metrics to measure the effectiveness of Safety Mechanisms include code coverage rate, SPFM (Single-point failure metric) and LFM (Latent failure metric). Especially in SPFM and LFM, if the specified value is not reached on the Fault Injection Simulation (using Gate Level) at the end of verification, it will cause iterations, which will cause a significant increase in time and cost compared to consumer LSIs.

View all articles ›