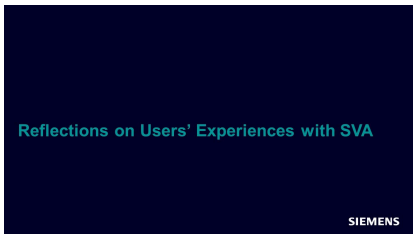
Reflections on Users' Experiences with SVA



In my years of contributions to the Verification Academy SystemVerilog Forum, I have seen trends in real users' difficulties in the application of assertions, the expression of the requirements, the angle of attacks for verification, the misunderstandings of how SVA works, and the confusion as to which SVA option to use.

SystemVerilog

Ben Cohen Last Updated Mar 2022

Assertions, Implies, Intersect, Operators, SystemVerilog, Throughout, Until, Vacuity, March 2022 - Volume 18 Issue 1, Verification Horizons, Beginner

Mark as completed

INTRODUCTION

In my years of contributions to the Verification Academy SystemVerilog Forum, I have seen trends in real users' difficulties in the application of assertions, the expression of the requirements, the angle of attacks for verification, the misunderstandings of how SVA works, and the confusion as to which SVA option to use.

In this first episode on the Verification Horizons, I am addressing two aspects of users' difficulties with SVA dealing with 1) expressing requirements for assertions; 2) SVA concepts concerning terminology, threads, and vacuity.

In future episodes, I'll address topics related to 1) the usage of these four relationship operators: throughout, until, intersect, implies; 2) workarounds of illegal SVA operations such as dynamic and range delays and repeats, dynamic $past of clock ranges, and stability of signals with no clocking events; 3) the verification of uniqueness in each attempted assertion; 4) the understanding of the sampling regions and their impact in expressing immediate and concurrent assertions and their action blocks; 5) the testing approach of assertions, and 6) the best dos and don'ts in writing assertions for formal verification.

EXPRESSING REQUIREMENTS

This is a GIGO issue. I see many users failing to express in English the definition of the requirements needed to write assertions, possibly because there is a misunderstanding of what the DUT is expected to perform, independently of the implementation; or because the users fail to understand the design requirements in light of the total picture; or because the users see requirements as defined by the implementation (i.e., code); or because there is no specification and the design is based on engineers' experiences with working on similar designs.

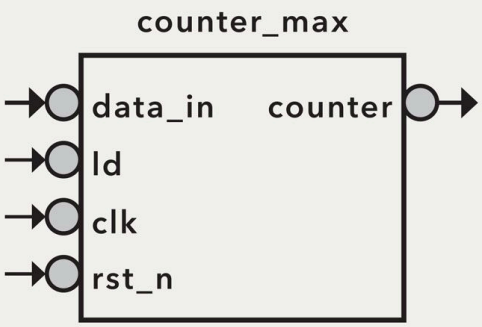
Defining requirements can be complex because there could be misunderstandings in what represents requirements. From my donated book[1] on this subject, I model by example how to express the requirements for a UART independently from the implementation.

When expressing requirements for verification, it is essential to consider all aspects, including restrictions and assumptions of what the chip should do and not do. When writing assertions, I recommend an English description of the requirements before

coding, as that would clarify the issues, document the code, and be useful in code reviews. Timing diagrams are also helpful when the requirements are complex. Breaking a large set of requirements into multiple smaller groups is important as that helps in the writing of assertions.

Those smaller sets can be considered functional partitioning, demonstrated in section 1.4.4, where a front-end interface interacts with a back-end processing block. I recommend testing those assertions separately from the design with a quick testbench using constrained-random tests. Consider this simple set of requirements for an unusual counter:

Table 1

* Loadable synchronous up-counter * Loadable counter, reset to MIN_COUNT if rst_n==0 * Min load value == MIN_COUNT (default ==2) * Max count value == MAX_COUNT (default ==9) * Holds the count when it reaches the maximum value * Must change value at least every 9 clocks * Increments if ld==1'b0 and the counter != MAX_ COUNT	

The above set of requirements, though somewhat complete, is still missing a lot of information and is incomplete.

Specifically:

- Are the values of MIN_COUNT and MAX_COUNT static or dynamic? If dynamic set, are there missing inputs for these values?
- How are error messages displayed in the design? Is there an output missing?
- Is the reset synchronous?

- What is the meaning of "Must change value at least every 9 clocks?" How is it to be interpreted? Does it mean that it has to wait for 9 clocks before any change, or can it change at any cycle, but the value cannot remain constant for more than 8 clock cycles?

I posted a model with assertions and a testbench for this counter[2]. For simplicity, I followed the UVM spirit, but not UVM, by including in the testbench the following class objects: data_item that represents the objects to be randomized (e.g., data_in, ld), transaction that represents the activity to be performed (e.g., READ, WRITE, RESET), and sequence that represents the randomization of the transactions and items (see link on VMM[3]).

SVA CONCEPTS

One of the most misunderstandings in the application of SVA is the concept of overlaps and threads in range repeats and range delays, and that leads to poorly written properties. I provided a model of such concepts in my paper *Understanding the SVA Engine Using the Fork-Join Model*[4]. A review of those SVA fundamentals and solution guidelines is provided below.

Terminology

Before getting into the topic of threads, a review of key terms when addressing SVA sequences, properties, and assertions is necessary because these terms are often misused. Let's "speak" SVA using proper words and grammar:

Match: A sequence is either matched or non-matched; that's it! It is not addressed as pass/success/fail/true/false, though many users consider a match a pass.

True/False: A property is either true or false. There is also the concept of vacuity, and a vacuously true property is considered 'true' in the sense that it is not 'false'. Consider the following property that has three property threads: a ##[1:3] b |-> c. For this property to succeed, we need at least one thread to succeed nonvacuously and no thread to fail. Thus, there can be vacuously true threads.

Pass/Fail/Active: An assertion either passes or fails or is active (*i.e., not yet completed with a determination of its fate*). There are two types of "passes": vacuous pass and nonvacuous pass. Nonvacuous passes are typically counted by tools and can thus be considered covered. A sequence that is interpreted as a property is never vacuous.

Covered: A property/sequence/expression is considered covered if it passes nonvacuously with the cover property, cover sequence, or cover statements. A covered

property is evaluated as strong. Tools provide statistics on passed assertions and assumptions, which can be counted as covered.

Thread: A thread is a search path for the property's success within an evaluation attempt that starts at the leading clocking event of the property. For example, a bus request that is acknowledged within 1 to 2 cycles has 2 threads: *request* followed by *acknowledge* in 1st cycle, or a request followed by *acknowledge* in 2nd cycles. A sequence is multi-threaded if it includes any of the following operators: and, intersect, or, range delay (e.g., ##[1:3] a), consecutive range repetition (e.g., a[*1:3]), and non-consecutive repetition (e.g., a[->1], a[=1]).

Attempt vs. thread: An attempt is a new evaluation of an assertion, and that assertion starts one or more threads. In some posts, I see people talk about attempts as threads interacting with each other; they don't! The threads from each successful attempt are separate from other launched threads from other attempts.

Threads: Range Delays in the Antecedent

Consider the following property:

```
$rose(a) ##[1:$] b |-> ##3 c; // antecedent has an infinite number of endpoints.
```

It states that at each endpoint of the antecedent, the variable c must be true three cycles later. The issue that arises with many users is that the antecedent is multithreaded, and that can cause unexpected errors. For a property to be true, **SVA requires that each of the threads of that antecedent with a range delay or a range repeat statement must be exercised with its appropriate consequent.**

The above property is equivalent to:

```
($rose(a) ##[1] b) or ($rose(a) ##[2] b) or ..($rose(a) ##[∞] b) |-> ##3 c;
```

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

```
($rose(a) ##[1] b |-> ##3 c) and // separate thread 1
($rose(a) ##[2] b |-> ##3 c) and // separate thread 2
...
($rose(a) ##[∞] b |-> ##3 c);// ∞ It is not a legal SVA syntax
```

This property can never be true because there are infinite threads. The Property can fail if any of its infinite threads fail.

SOLUTION: Use the goto operator [->1] or the **first_match()** method in the antecedent, provided that the user only wants the first occurrence of a matched

antecedent to be evaluated with the consequent; in most designs, this is the typical case.

```
$rose(a) ##[1:5] b |-> ##3 c; // Avoid this
first_match($rose(a) ##[1:5] b) |-> ##3 c;  // Do this for infinite range
$rose(a) ##[1:$] b   |-> ##3 c; // Avoid this
$rose(a) ##1 b[->1] |-> ##3 c; // Do this for infinite range

Note: b[->1] is equivalent to: !b[*0:$] ##1 b
```

Do not use the goto as the first term of an antecedent in an assertion that triggers at every clocking event *(seen in many posts)*. You may use it in an assertion embedded as a single attempt in an initial statement or within a property that runs concurrently with other properties (e.g., P1 **and** P2; P1 **implies** P2).

For example:

```
ap_BAD: assert property(@ (posedge clk) a[->1]  |-> b); // Use a |-> b
// At every clocking event you create a new multithreaded antecedent that ends when a==1
initial ap_OK:  assert property(@ (posedge clk) a[->1]  |-> b);
// Not for formal verification
//This assertion has a single attempt. The consequent is evaluated when a==1.
```

Threads: Ranges in Repeats Have the Sequential "or" Operator:

The same principles described above apply to the repeats on ranges.

```
a[*1:$] is equivalent to a[*1] or a[*2] or .. a[*n].
SOLUTION: Use first_match()
$rose(a) ##1 b[*1:5] ##1 c |-> ##3 d; // Avoid this
first_match($rose(a) ##1 b[*1:5] ##1 c) |-> ##3 d;  // DO this
```

Vacuity: Multiple Properties Each with an Antecedent

Consider the following property with a nested Implications style that I have seen many users express:

```
sequence1 |-> sequence2 |-> sequence3  |-> sequence4;
                  <—————————property34—————————>
      <—————————————————property2_34—————————>
<—————————————————property1__2_34—————————>
```

The issue with this style is that each non-matched antecedent causes that property to be vacuous, and thus the whole property becomes vacuous. For example, if sequence3 is a non-match, then property1__2_34 is vacuous.

SOLUTION: A property written in this style appears too complex to be written as a single property. In general, it is best to write multiple smaller assertions, with each

addressing a small portion of the requirements.

For example:

```
Prop-1: sequence1 |-> sequence2
Prop-2: sequence2 |-> sequence3
Prop-3: sequence3 |-> sequence4;
```

Consider the requirements, perhaps this is what is intended:

```
sequence1 ##0 sequence2 ##0 sequence3 |-> sequence4;
sequence1 |-> sequence2 ##0  sequence3  ##0 sequence4;
```

CONCLUSIONS

The many posts on assertions, many with unusual requirements, clearly indicate that SVA is now well embedded into our design verification process. The biggest hurdles that users experience are expressing the requirements that are the baselines to start SVA. Other issues are the lack of understanding of how SVA works, partitioning the assertions to simplify them, and selecting the appropriate constructs.

Those posts also demonstrate the influx of junior engineers who are very avid in learning and are struggling with the language (SVA and SystemVerilog). This article attempted to address, by example, many of those issues. More articles to follow…

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

END NOTES

1. Section 1.1: Component Design by Example
2. Model and verification for a complex counter
   a. https://verificationacademy.com/forums/systemverilog/error-suppressible-vlog-7061-alwaysff-modelsim
   b. https://verificationacademy.com/forums/uvm/tb-architecture
   c. https://verificationacademy.com/forums/systemverilog/systemverilog-or-uvm#reply-10220
3. A Pragmatic Approach to VMM Adoption, a SystemVerilog Framework, Donating book as it explains the transaction-based constrained-random testing concepts.

a. SystemVerilog.us/VMM/VMM_code_release071806.tar
b. SystemVerilog.us/VMM/VMM_pdf_release070506.zip
4. Understanding the SVA Engine Using the Fork-Join Model

Back to Top

March 2022 Articles

NVMe-oF – Simple, Invisible Fabric to Cloud Storage

Verificati... Article

In this era of digitalization, we can manage most of our personal stuff online, such as handling bank transactions, ordering clothes, and booking cab rides. The COVID-19 pandemic has pushed us even closer to digitalization. We are now ordering groceries online and entertaining ourselves through the plethora of content available on various streaming platforms. Inevitably, there has been significant growth in the amount of data available online and the number of users consuming it.

Getting to Know Visualizer - Part I

Debug Article

The Visualizer Debug Environment is the user interface to debug, analyze and verify all our functional verification tools.

Visualizer is first and foremost a waveform debugger. It also includes source code debug, transaction debug, C debug, driver tracing, X tracing, schematics, glitch debug, low power debug, and coverage analysis and coverage debug – all supporting Verilog, SystemVerilog, VHDL, System C, and C/C++.

Enabling Model-Based Design for DO-254 Certification Co...

Function... Article

Engineers can use Model-Based Design for requirements analysis, algorithm design, automatic HDL code generation, and verification to produce airborne electronic hardware that adheres to the DO-254 standard. The proposed Model-Based Design approach for DO-254 combines tools from MathWorks® and Siemens EDA for both design and verification. This workflow supports development phases from concept through implementation, streamlining development, and reducing costs.

How Do You "Qualify" Tools for DO-254 Programs?

Function... Article

Tools used in the design and verification of electronics have played a massive role in the dramatic evolution of these devices over the past few decades. After all, there is a limit to the amount of work and detail that even a good aerospace engineer can handle, but add the use of tools, and the sky (pun intended) is the limit.

Reflections on Users' Experiences with SVA

SystemVe... Article

In my years of contributions to the Verification Academy SystemVerilog Forum, I have seen trends in real users' difficulties in the application of assertions, the expression of the requirements, the angle of attacks for verification, the misunderstandings of how SVA works, and the confusion as to which SVA option to use.

A Faster Approach to Co-Simulation Using Questa and VPI

Simulation Article

Co-simulating systems, including RTL and software, may often require excessive computational times if a cycle-accurate CPU model is used. However, many co-simulation exercises do not necessarily require precise CPU models and may benefit from the solution proposed here.

This article presents a not-widely-used method of co-simulation that doesn't need a cycle-accurate CPU simulation model and reduces simulation time while still allowing functional testing of software and RTL.

UVVM – VHDL Verification Methodology for Faster and Better... VHDL ... Article

Verification takes half of a typical FPGA project's development time. It is possible to significantly reduce this time with only minor adjustments and no extra cost while dramatically increasing the ability to reuse testbench components.

An FPGA design's architecture – from the top to the microarchitecture – is critical for both the FPGA quality and the development time. The same is true of the testbench.

View all articles ›