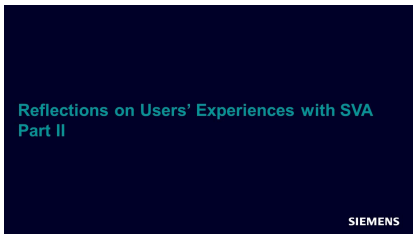
Reflections on Users' Experiences with SVA - Part II



During my years of contributions to the Verification Academy SystemVerilog Forum, I have seen many trends in real users' difficulties in the application of assertions, and misunderstandings of how SVA works. In Part 1 of this article, I addressed the difficulties in expressing requirements for assertions, and clarified some critical SVA concepts concerning terminology, threads, and vacuity.

SystemVerilog

Ben Cohen Last Updated Jul 2022

Assertions, Implies, Intersect, Operators, Throughout, Until, SystemVerilog, July 2022 - Volume 18 Issue 2, Verification Horizons, Beginner

Mark as completed

INTRODUCTION

During my years of contributions to the Verification Academy SystemVerilog Forum, I have seen many trends in real users' difficulties in the application of assertions, and misunderstandings of how SVA works. In Part 1 of this article, I addressed the difficulties in expressing requirements for assertions, and clarified some critical SVA concepts concerning terminology, threads, and vacuity.

In Part 2 of this article, I will address the usage of these four relationship operators: throughout, until, intersect, implies.

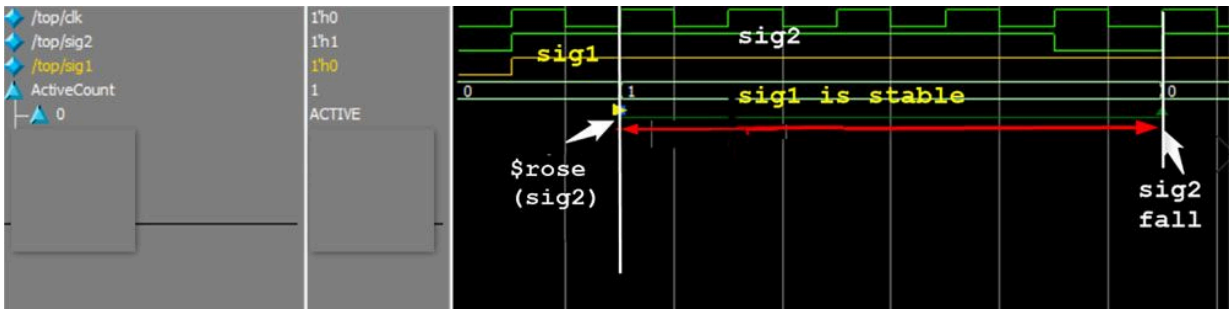
OPERATORS: THROUGHOUT, UNTIL, INTERSECT, IMPLIES

Sequences that use the keywords throughout and intersect describe the relationship between sequences. The until (**s_until, until_with, s_until_with**) describes a tight relationship between properties. The implies describes a loose relationship between two properties. However, if those properties are sequences, then we can say that they too describe relationships between those sequences. Choosing one of these keywords when handling sequences has caused difficulties with users. For clarity, I'll briefly describe each of them and provide recommendations.

Consider the following requirement from an actual user (Figure 1):

Upon a rising of ***sig2, sig1*** must be continuously stable until (and including) the fall of ***sig2***. In other words, **sig1** does not change its value starting from the cycle after the rise of ***sig2*** until the fall of **sig2.** During that fall cycle, ***sig1*** is still stable.

Figure 1 - Requirement Diagram (link[1])



Links to full solutions using the until_with, intersect, and throughout are included in the footnotes[2] and are discussed below:

```
ap_sig1_stable_intersect_sig2 : assert property(@(posedge clk)
     $rose(sig2) |=> ($stable(sig1)[*0:$] ) intersect  !sig2[->1]);
      // !sig2[->1] is same as (sig2[*0:$] ##1 !sig2)

ap_until_sig1_stable_until_sig2: assert property(@(posedge clk)
     $rose(sig2) |=> $stable(sig1) until_with !sig2);
      //  $rose(sig2) ##1 |-> $stable(sig1) until_with !sig2); // Also OK

ap_sig1_stable_throughout_sig2_B : assert property(@(posedge clk)
     $rose(sig2) |=>  $stable(sig1) throughout (sig2[*0:$] ##1 !sig2));
      // $rose(sig2) |=>  $stable(sig1) throughout !sig2[->1] // OK
```

Intersect

The *intersect* is a length-matching sequence conjunction. It is a sequence operator used when both operands are expected to match, and the endpoints of the sequences end in the same cycle. A sequence intersect operator between two sequences starts two concurrent threads, one for each sequence; each sequence, in turn, can be multi-threaded.

The syntax for this sequence expression is: *sequence_expr* intersect *sequence_expr.* I see the intersect construct as more hardware-oriented. It emulates a gate or a timing envelope for another sequence to be a match; it is thus very explicit about the definition of each of the two sequences. Unlike the other two constructs (until and throughout), the sequences have an underlying behavior.

With the intersect, that gating envelope where ***sig1*** must be continuously stable until (and including) the fall of ***sig2*** is represented by the envelope *!sig2[->1]).* This specifies a sequence that ends at the first occurrence of the expression *!sig2.* The assertion is then:

```
$rose(sig2) |=> ($stable(sig1)[*1:$] ) intersect !sig2[->1]);
```

Looking for the coincidence of one matched sequence from the multi-threaded *$stable(sig1)[*1:$]* and one matched sequence from the multi-threaded sequence *!sig2[->1].*

On a separate but related note, another interesting application of the intersect is demonstrated in this assertion for a sequence *(a valid pulse)* completely enclosed within another sequence *(a busy pulse),* (link to code[3]).

```
ap_busy_valid: assert property (@(posedge clk)
     $rose(busy)  |-> ##1  //
```

```
$rose(valid)[->1] ##1 $fell(valid)[->1] intersect  $fell(busy)[=1] );
```

Until, until_with

The syntax for this property expression is:

```
property_expr1 until property_expr2 | property_expr1 until_with property_expr2
```

An *until* property of one of the overlapping forms (i.e., until, ***until_with***) evaluates to true if *property_expr1* evaluates to true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing *until*, and including a clock tick at which property_expr2 evaluates to true. Thus, the difference between non-overlapping and overlapping is that the ***until_with*** form requires that the first expression holds through the terminating condition. Specifically, *(p **until_with** q)* is equivalent to *((p until (p **and** q)).*

The ***until_with*** is more English-like in interpretation, but IMHO, it requires mental effort to visualize its full meaning. Specifically, the **$stable**(sig1) **until_with !sig2** means that **sig1** is stable at every cycle, including its end condition, which is the first occurrence of **!sig2**. Thus, the (**$stable**(sig1)) until .. has an implicit repeat operation of the expression **$stable**(sig1).

```
ap_until_sig1_stable_until_sig2: assert property(@(posedge clk)
    $rose(sig2) |=> $stable(sig1) until_with !sig2);
```

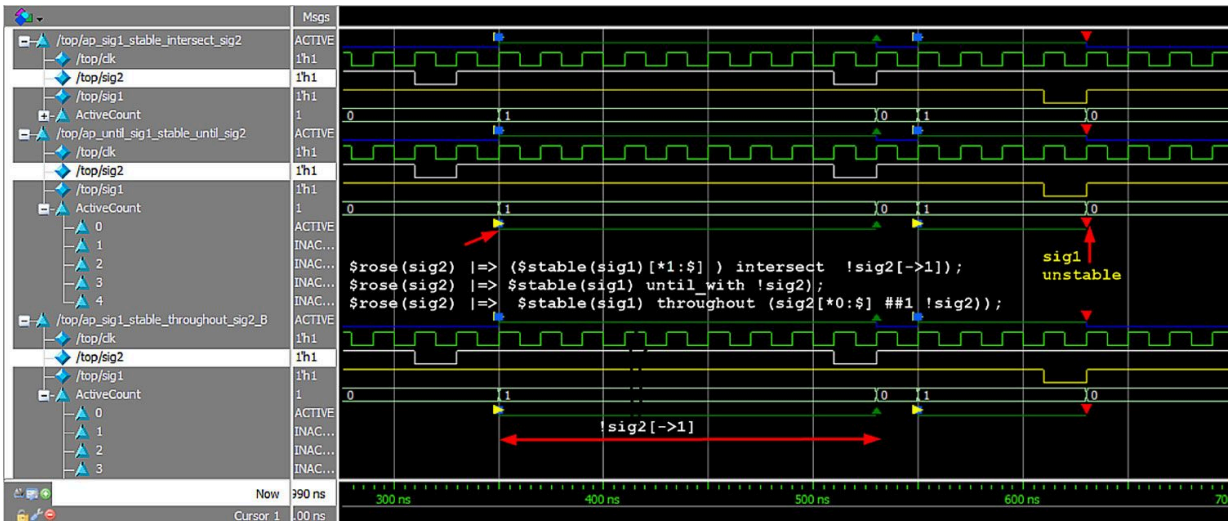
Throughout

The syntax for this sequence is: *expression_or_dist* throughout *sequence_expr*. The throughout operator specifies that a signal (expression) must hold throughout a sequence. (b throughout R) is equivalent to (b **[*0:$]** ***intersect*** R)

This construct is also English-like, but is clearer (IMHO) than the ***until_with*** because there is a clearer definition of the envelope in which the expression is continuously true. The expression (the **$stable**(sig1) throughout ..) has an implicit repeat operation of sig1.

```
ap_sig1_stable_throughout_sig2_B : assert property(@(posedge clk)
     $rose(sig2) |=>
        $stable(sig1) throughout (sig2[*0:$] ##1 !sig2));
```

Figure 2



The "implies" Though not often used, the *implies* property operator has merit in many situations. The *implies* has the form: *property_expr1* implies *property_expr2.* It evaluates to true if either *property_expr1* evaluates to false or *property_expr2* evaluates to true. It is equivalent to: *(not property_expr1) or (property_expr2).*

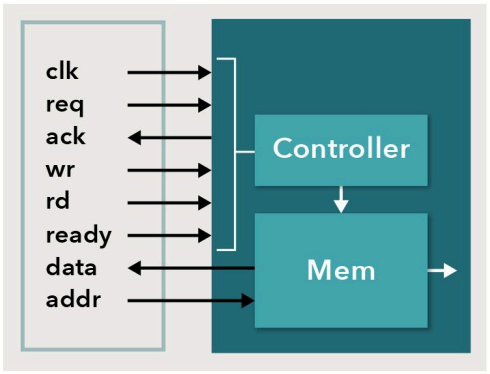
There are three differences between the implies and the sequential implication operators:

implies	|-> |=>
1. Antecedent and consequent are properties.	1. Antecedent is a sequence, consequent is a property.
2. Antecedent and consequent start at the same evaluation time and may end at different cycles.	2. Consequent property starts evaluating at each matching endpoint of the antecedent.
3. Local variables of one property are not visible in another one.	3. Local variables of antecedent flow through to the consequent property

The only similarity between the two is vacuity. If the antecedent of the implication operator is a no-match or if the LHS property of the *implies* is false, then the whole

property is vacuous. You can also view the LHS property of the *implies* as an assumption. The *implies* can be interpreted as assuming that p1 holds, property p2 must be non-vacuously or vacuously true.

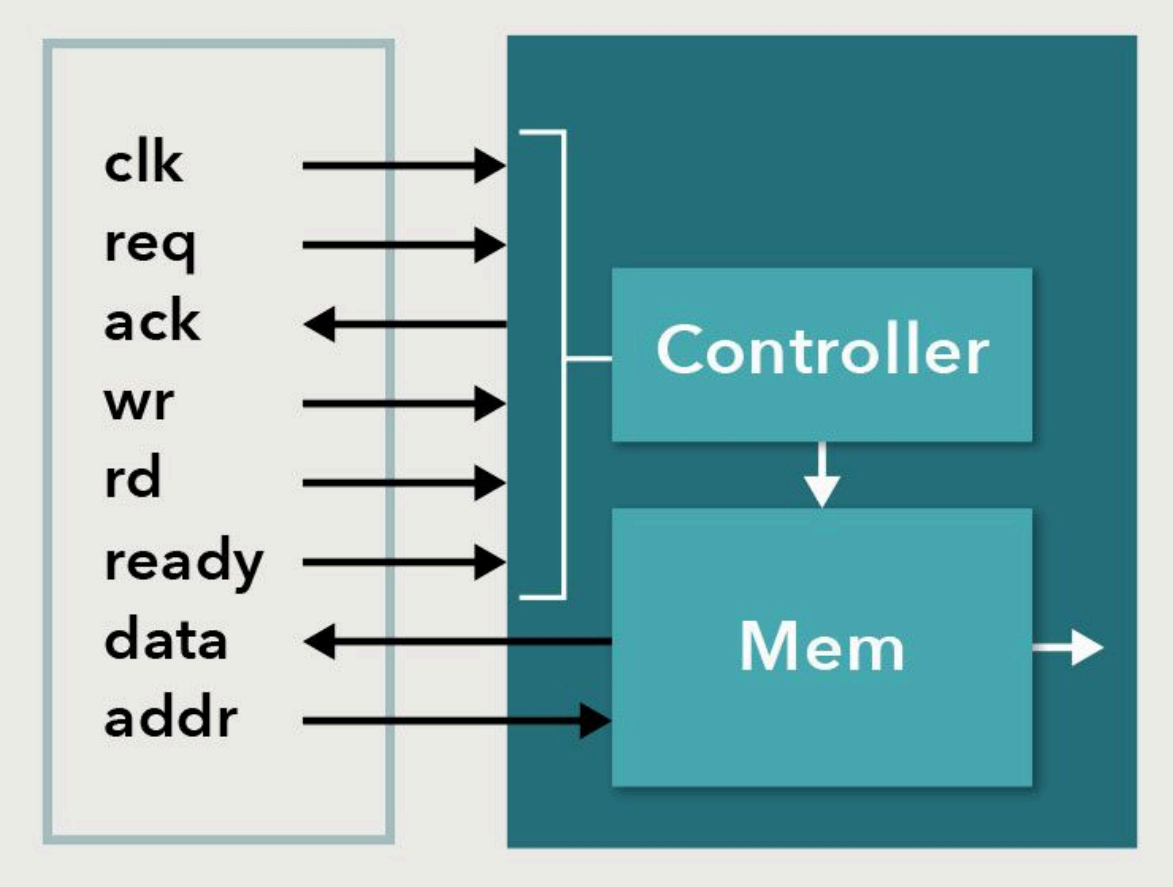
This operator is helpful when there is a relationship between the properties, but each addresses a different design characteristic, even using other signals. Consider the following model that has a front-end interface and a back-end process. Each of these processes can be described separately.

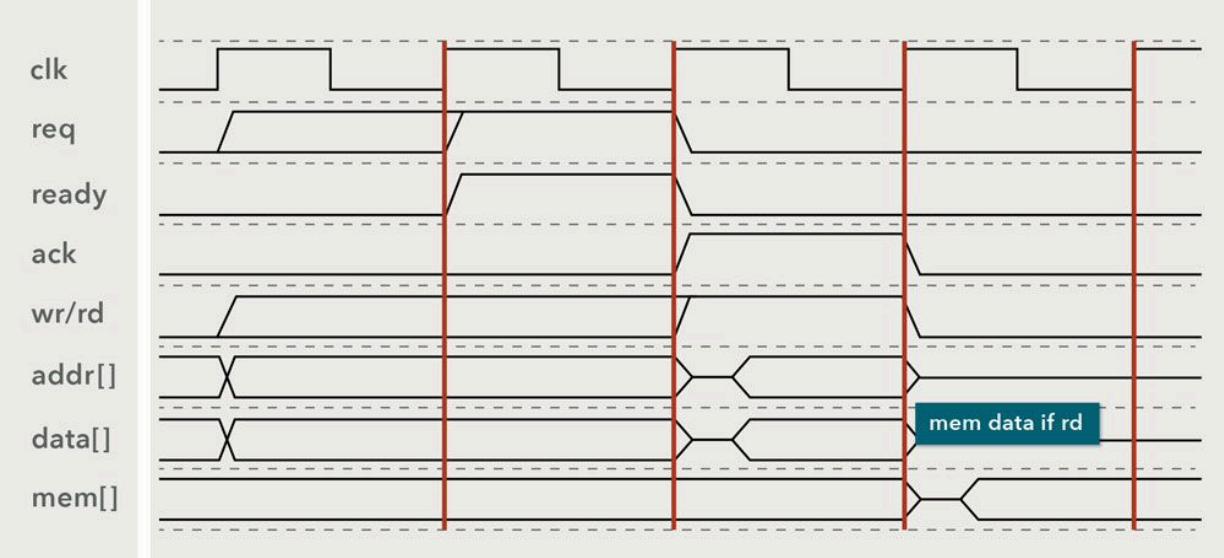


The front-end process handles the handshaking interface for a read or a write transaction. The back-end process addresses the memory storage. If the front-end process occurs, then the backend process must also occur.

Describing these two processes separately provides their focused definitions. The `implies` operator provides the relationships between these properties.

The timing diagram and the body of the properties are shown on the following page. The complete model and simulation waveforms are available at this link[iv].





FRONT-END PROPERTIES p_interface: (req && ready|-> ##1 ack); ap_no_rd_wr: not(rd && wr && ack));	Handshaking Front-end property The **rd/wr** signals are unimportant Restriction on the interface
BACK-END PROPERTIES p_wr2mem: // *v_addr, v_data are local variables* req ##1 (wr && ack, v_addr=addr, v_data=data) |-> ##1 mem[v_addr]==v_data; p_read_mem: bit[0:9] v_addr; req ##1 (rd && ack, v_addr=addr) |-> ##1 data==mem[v_addr]	Back-end property starts in the same cycle as the handshake but handles the memory write aspect of the design. Back-end property starts in the same cycle as the handshake but handles the memory read aspect of the design.
FRONT-END implies BACK-END p_wr2mem_implies: // *write to mem* p_interface implies p_wr2mem; ap_read_mem_implies: // *read from mem* p_interface implies p_read_mem);	Implications relating the front-end and back-end properties using the ***implies*** property operator. Note: The *p_interface* property is reused. RHS and LHS start at the same cycle.

FRONT-END |-> BACK-END
p_reqackwr_mem: // *wr request to mem load*
req && ready ##1
(wr, v_addr=addr, v_data=data)|->
ack ##1 mem[v_addr]==v_data;
 	
p_reqackrd_mem: // *rd request to mem read*
req && ready ##1
(rd, v_addr=addr)|->
ack ##1 data==mem[v_addr];
	Implications relating the front-end and back-end properties using the sequential implication operator.
	Note: For each rd and wr, these properties combine the front-end and back-end in a single property instead of separating the front-end and back-end into separate properties and then combining them. LHS starts when RHS ends.

GUIDELINE: I generally prefer the *intersect* construct because it represents the intent. The throughout is my next alternative, but the reader needs to understand the underlying operations. The *until*, and until_with constructs are my next alternatives. In all cases, the reader must be knowledgeable about the meaning of these constructs. There are contrarian opinions on this; a LinkedIn survey of usage yielded a surprising conclusion with 39% in favor of the throughout, 36% preferring the *implies*, 14% for the *until*, and 11% for the *intersect*.

The *implies* is useful when describing two separate independent properties, each dealing with their aspect of the requirements. Yet, they are related because if the LHS property is true, it *implies* that the RHS property must also be true. Using this style allows the tuning of the properties separately. All properties used in the *implies* need to be exercised in separate assertions because they each address smaller pieces of the combined property.

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:

- 1800'2017: IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language
- SystemVerilog Assertions Handbook, 4th Edition for Dynamic and Formal Verification, Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, and Lisa Piper
- https://verificationacademy.com/forums/systemverilog

END NOTES

- http://SystemVerilog.us/fv/reqsig1.png
- Assertion models and simulation results:
  - http://SystemVerilog.us/fv/stable_sig1.sv,
  - http://SystemVerilog.us/fv/stable_sig1.png
- https://verificationacademy.com/forums/systemverilog/writing-sequence-within-sequence#reply-102275
- http://SystemVerilog.us/vf/implies_ack.sv:
  - http://SystemVerilog.us/vf/implies_slide1.png
  - http://SystemVerilog.us/vf/implies_slide2.png

Back to Top

July 2022 Articles

Back to the Future with Formal Property Checking

Formal V... Article

Back in 2010, I decided that instead of documenting a specific instance of applying formal property checking on a particular design, I would step back and look at the formal property checking process holistically. The goal was to define a set of repeatable steps that could be applied to any design. Fast forward to today, where I update the original by keeping the content that is still relevant and augmenting it with the new philosophies, methodologies, and technologies that have evolved since.

Getting to Know Visualizer - Part II

Debug Article

Welcome to part 2 of our overview of the Visualizer Debug Environment, the user interface to debug, analyze and verify all our Siemens functional verification tools.

As we saw in part 1, Visualizer is first and foremost a waveform debugger, with a host of other powerful debug capabilities also provided and supporting Verilog, SystemVerilog, VHDL, System C and C/C++. In this part of the article, we'll look at driver tracing, X tracing, schematics, glitch debug, low power debug and more.

The Democratization of Digital Methodologies for AMS Verificati...

A... Article

A mixed-signal design is a combination of tightly interlaced analog and digital circuitry. Next-generation automotive, imaging, IoT, 5G, computing, and storage markets are driving the strong demand for increasing mixed-signal content in modern systems on chips (SoCs). There are two critical reasons for this trend.

Unblocking the Full Potential of SSDs Using Zoned and Key...

Verificati... Article

Since the advent of digitalization, there has been an exponential growth in the volume of data. With this boost in the amount of data, hard disk drives (HDDs) could not sustain the data transfer rates, leading to bottlenecks in data access. Solid state drives (SSDs) have come to the forefront as a promising solution to our modern-day storage demands. SSDs are constantly evolving with upgrades of their critical components to provide high access speeds.

Bringing 5G NR Radio Frame Generation and Analysis to the ...

Accele... Article

Booming worldwide development activities for 5G NR create an increasing need for reference data and signal analysis. Recently launched Veloce X-STEP IQ Toolset (IQT) provides 3GPP compliant test data for radio unit (RU) testing needs. IQT workflow integrates with Siemens EDA tools such as Questa, Veloce and X-STEP. Being a standalone software solution, it is a cost-efficient solution to be installed in multiple workstations.

Speeding OTN Verification Using Emulation

Accelerat... Article

In today's connected world, bandwidth requirements have shot up drastically due to the exponential growth in data communication. Optical Transfer Networks (OTN) today are the backbone of the tremendous amount of worldwide internet traffic. The ITU-T standards committee developed the OTN standard to address this data explosion with reliable infrastructure and low transmission costs in the optical world.

Reflections on Users' Experiences with SVA - Part II

SystemVe... Article

During my years of contributions to the Verification Academy SystemVerilog Forum, I have seen many trends in real users' difficulties in the application of assertions, and misunderstandings of how SVA works. In Part 1 of this article, I addressed the difficulties in expressing requirements for assertions, and clarified some critical SVA concepts concerning terminology, threads, and vacuity.

Easy Testbench Speedups

SystemVe... Article

As a Doulos 'techie', I train over 100 engineers in SystemVerilog and UVM each year. I do believe quite soundly, that the effort of simulation verification is an art, supported by the language. So, regardless of the language, I have a ready list of useful testbench coding strategies to achieve faster regression CPU cycle execution. This means more regression tests executed in the same amount of 'wall-clock' time!

Functional Safety Verification Challenges for Automotive ICs

Function... Article

In a semiconductor world, functional safety is all about data storage and data movement through the system. Electrical or magnetic interference inside hardware systems can cause a single bit to flip to the opposite state spontaneously. And this is a typical case for random failure, which we desperately need to analyze and see its effects on the functional behavior of the system we are verifying.

The Path to a Safety Mechanism on an Unsafe PCIe® Sub-...

Function... Article

This article illustrates the implementation of Safety Mechanisms on an unsafe PCIe® sub-module and demonstrates the use of Siemens EDA Austemper tools to generate Alarms for fault list detection and ensure Safety using a Duplication Mechanism.

View all articles ›