(intersect) vs (throughout, until, until_with, within)

Ben Cohen 10/26/2024

SystemVerilog.us

1.0 Introduction

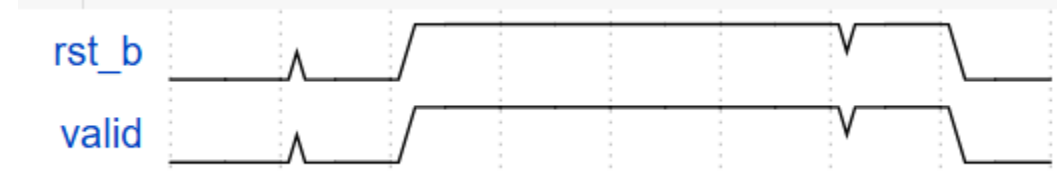
SystemVerilog Assertions (SVA) offers an intuitive, English-like syntax to express sequence relationships, including the **'throughout'**, **'until'**, **'until_with',** and **'within'** operators. The **'throughout'** and **'within'** operators are based on the **'intersect'** operator.

This paper is a follow-up to my *Verification Horizons July 2022* paper "*Reflections on Users' Experiences with SVA, part 2*[i]*".* I revisit these operators' definitions and provide examples of users' misinterpretations. These misunderstandings are often caused by inherent misconceptions arising from the English-like syntax, which can be misleading despite being clearly defined in the IEEE 1800 standard.
I illustrate how the natural language appearance of these constructs can lead to incorrect assumptions about their true behavior and implementation.

Throughout this paper, I use a consistent example to demonstrate the application of these operators. This approach provides a clear and coherent illustration of how each operator functions in practice, allowing for direct comparisons and highlighting the nuances between them. By examining the same scenario through different operators, I aim to enhance understanding and reveal potential pitfalls in their usage.

Example requirements: When 'rst_b' goes high, signal 'valid' should also go high and stay high as long as 'rst_b' is high. The 'valid' should go low in the same cycle that 'rst_b' goes low. In the cycles following that new low, if 'rst_b' is low then 'valid' can be high or low[ii].

```
rst_b   0  0  1  1  1 ... 1  1  0  0  0  0  0
valid   0  0  1  1  1 ... 1  1  0  0  1  1  0
```



Note: One could implement this assertion with two simple assertions such as:

```
ap_simple0: assert property(@(posedge clk) rst_b |-> valid);
ap_simple1: assert property(@(posedge clk) $fell(rst_b) |-> !valid);
```

It could also be expressed in one assertion such as

```
ap_simple1: assert property(@(posedge clk) $rose(rst_b) |->
          (valid && rst_b[*1:$] ##1 !rst_b)  ##0 !valid;
```

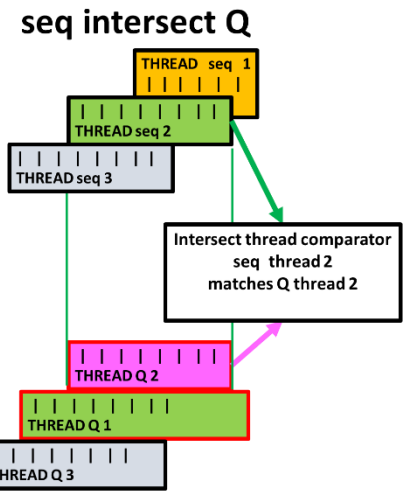
However, I'll use the other constructs to demonstrate the more complex operators. I'll also concentrate on the consequent sequence rather than the full property. This approach addresses a broader range of requirements and showcases these operators' versatility and confusion.

2.0 The 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.

The sequence '**intersect'** operator, when applied between two sequences, initiates two concurrent threads, one for each sequence. Each of these sequences, in turn, has the potential to be multi-threaded itself.

The syntax for this sequence expression is:
`sequence_expr` **`intersect`** `sequence_expr`



A crucial aspect of using the '**intersect'** operator is the clear definition of the signatures for both sequences that need to be matched, from their starting points to their endpoints. The operator functions explicitly based on these defined signatures, without any hidden or implied intentions. This clarity in definition ensures that the behavior of the '**intersect'** operation is transparent and predictable.
Using the template example where signal '`valid`' tracks 'rst_b', we can write the consequent sequence as:

```
(valid[*1:$] ##1 !valid) intersect (rst_b[*1:$] ##1 !(rst_b))
(!valid[->1]           ) intersect (!rst_b[->1])  // equivalent version
```

3.0 The 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.
Per 1800'2023, the construct 'exp **throughout** seq' is an abbreviation for the following:

```
      (exp) [*0:$] intersect seq
```

To implement the requirements for the example using the '**throughout'**, a verification engineer incorrectly wrote this sequence as:

```
(valid throughout  (rst_b[*1:$] ##1 $fell(rst_b))) ##0 !valid;  // BAD
// Note: the $fell(rst_b) can be replaced with !rst_b as it is more efficient than $fall
```

The engineer aimed to define a scenario where `'valid'` remains true throughout the sequence while `'rst_b'` is high, up until `'rst_b'` falls to zero. This sequence considers only the `'rst_b'` signal. Additionally, the engineer included a condition for `'valid'` to be false in the final cycle when `'rst_b'` falls. However, this approach results in an inherent contradiction: `'valid'` is required to be both true (due to the **`'throughout'`** operator) and false (in the final cycle) simultaneously. Consequently, this sequence will always fail to match, as it's impossible for `'valid'` to satisfy both conditions in the same cycle.
Another possibility for the error is that the engineer failed to explicitly visualize the cycle extent of `'valid'`, necessitating a mental translation. This mental imagery encompassed the duration of a matching

thread with the sequence `(rst_b[*1:$] ##1 $fell(rst_b))`, but then erroneously added `##0 !valid`, because `'valid'` must be zero in the final cycle. While this reasoning is convoluted, the attempt to intersect the two sequences resulted in an awkward construction, partly because the left-hand side of the **`'throughout'`** operator must be an expression.

In another situation, an engineer wrote the following sequence for signal `a`==1 as long as signal `b`==1:

```
siga throughout sigb
```

In this example, the engineer misinterpreted the **`'throughout'`** operator by applying an English language understanding rather than adhering to its IEEE 1800 definition. The word **`'throughout'`** in everyday English implies a continuous duration, which led to the logical, but incorrect, assumption that `(b throughout c)` meant "as long as b==1, then c==1"; this is incorrect!

These examples highlight how the English-like syntax, while designed for readability, can sometimes lead to incorrect assumptions about the underlying operations.

4.0 The until, until_with

4.1 Understanding the operator

The until is not a sequence operator, but a property operator. The syntax for this property expression is:

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

Per 1800'2023: "An until property of the non-overlapping form (e.g., **`until, s_until`**) 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 at least one tick before a clock tick where *property_expr2* evaluates to true". The "at least" indicates that `property_expr1` must be true for a minimum of one clock cycle before `property_expr2` becomes true, but it can be true for more cycles.

"An until property of one of the overlapping forms (i.e., **`until_with, s_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".

Though the **'until'** and **'until_with'** operators are property operators, they are frequently employed as if they were sequence operators and a substitute for the '**throughout'** operator. In these constructs, the left-hand side (LHS) and right-hand side (RHS) are typically defined as expressions and sequences. The simulation below compares the evaluations of properties that use sequences for the RHS and LHS of the operator.

```
ap0_until: assert property(@ (posedge clk) $rose(go) |->
       (a ##1 b ##1 c) until (d ##1 e) ) p0++; else f0++;      // LHS_seq until RHS_seq
ap1_until_with: assert property(@ (posedge clk) $rose(go) |->
       (a ##1 b ##1 c) until_with (d ##1 e) ) p1++; else f1++; // LHS_seq until_with RHS_seq
```

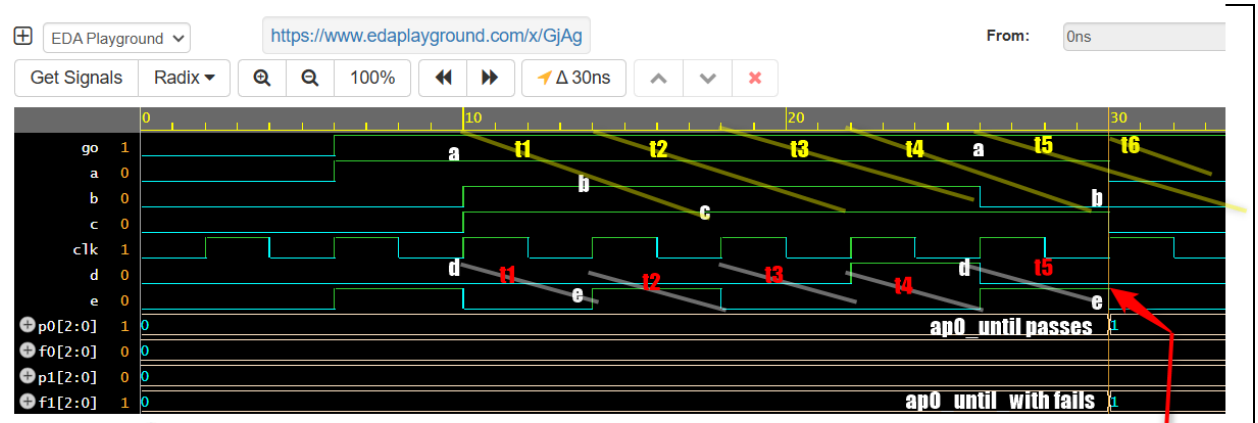
The simulator initiates both the left-hand side (LHS) and right-hand side (RHS) sequences at every clock cycle (e.g., `posedge clk`), creating multiple concurrent threads. The simulation example below illustrates this concept. Note the following:

- The LHS_seq (e.g., `a ##1 b ##1 c`) initiates 6 threads, with the 1st one starting at `t10`, the 2nd at `t14`, and the 6th thread starting at `t30`.
- The RHS_seq (e.g., `d ##1 e`) also has 6 threads and matches at the 6th thread start at t30.

For the property to evaluate as true, every launched LHS thread must end with a match; there is an exception on the last thread for the **`'until'`**. This thread generation continues until the RHS_seq has a match, which serves as the "terminating condition"; that occurs at the 6th thread start. Per 1800, the LHS thread that started one cycle before the terminating condition (*i.e.*, *LHS thread 5*) is not necessary for the evaluation of the **`'until'`** operator, but it is crucial for the **`'until_with'`** operator. Consequently:

1. The property `(a ##1 b ##1 c)` **`until`** `(d ##1 e)` evaluates to TRUE at t30 because each of the LHS_seq threads before the terminating condition (thread 1 to 4) ended with a match. Thread t5 started one cycle before the terminating condition and is thus ignored.
2. The property `(a ##1 b ##1 c)` **`until_with`** `(d ##1 e)` evaluates to FALSE at t30 because the thread t5 that started one cycle before the terminating condition is considered in the evaluation. If thread t5 of the LHS_seq was a match then `ap1_until_with` would be a pass at its termination at t36 https://www.edaplayground.com/x/FQ9M code, https://www.edaplayground.com/w/x/RDv wave

   The simulation results where the LHS is not a match at every cycle is shown below.
   https://www.edaplayground.com/x/GjAg code https://www.edaplayground.com/w/x/XNn wave



Terminating condition
In thread 6, (d ##1 e)
is a match

In the evaluations of threads t6 at t30:
1) LHS thread 5 is NO match because b==0
2) Because LHS t5 started 1 cycle before the terminating condition
The 'until' ignores this thread
4) The 'until_with' uses this last LHS t5 thread and thus fails at 30

```
ap0_until:      assert property(@ (posedge clk) $rose(go) |->
                                (a ##1 b ##1 c) until (d ##1 e) ) p0++; else f0++;
ap1_until_with: assert property(@ (posedge clk) $rose(go) |->
                                (a ##1 b ##1 c) until_with (d ##1 e) ) p1++; else f1++;
```

```
// Key aspects of the testbench:
  bit clk, a, b, c, d, e, go;
  bit[2:0] p0, p1, f0, f1;
  bit[2:0] abc [0:8]={0, 3'b100, 3'b111, 3'b111, 3'b111,3'b111, 3'b101, 0, 0};
  bit[1:0] de [0:8]={0, 2'b01, 2'b00, 2'b01, 2'b00,2'b10, 2'b01, 0, 0};
  initial begin    // The test vector generation
    @(posedge clk)
    go<=0;
    for (int i=1; i<=9; i++ ) begin
      @(posedge clk) begin
          go<=1;
            {a, b,c} <= abc[i];
            {d,e} <= de[i];
        end
    end
    #20;
    $finish;
  end
```

4.2 Model Example

The model example implements the above sequence requirement *(signal* `valid` *must hold while* `rst_b` is true. The simulation results are shown in section 7.

```
  valid until ($fell(rst_b) && !valid) // Appears correct, but is NOT correct!
   // at termination, when the RHS is true, the until does not care about the value of the LHS
```

This solution using the **`'until'`** operator is flawed because it doesn't accurately capture the intended behavior. The issue arises from the possibility that `'rst_b'` could fall before `'valid'` becomes false. This means the RHS property of the **`'until'`** might only evaluate to true when it finds a cycle where both `($fell(rst_b) && !valid)` are true simultaneously, which could occur on the second, third, or even the nth instance of `'rst_b'` falling. This behavior doesn't align with the original intent of the assertion, which was to capture the first instance of `'rst_b'` falling and `'valid'` becoming false.

This misalignment between the intended behavior and the actual implementation highlights the importance of carefully considering the temporal aspects and precise semantics of SystemVerilog operators when constructing assertions.

Upon considering the use of the **`'until_with'`** operator, we encounter another significant issue.

```
  valid until_with ($fell(rst_b) && !valid)
```

At the point of termination, we face a contradiction:
- The left-hand side (LHS) requires valid==1
- The right-hand side (RHS) requires valid==0

These conflicting requirements cannot be simultaneously satisfied in the same cycle. As a result, we find ourselves in a predicament aptly summarized by the phrase: "Houston, we've got a problem."

This contradiction highlights the need to fully understand the implied values of signals when using the **`'until'`** and **`'until_with'`** operators. It underscores the importance of carefully considering these operators' behavior in your specific scenarios.

Note: The '**until/until_with'** is a property operator, thus you cannot concatenate a sequence to a property. The following is illegal, though it looks like it would solve the "Houston problem".

```
(valid until $fell(rst_b)) ##0 !valid  // Illegal
 <------property ------>  <sequence->
```

5.0 The within

The sequence containment '**within'** specifies a sequence occurring within another sequence.

```
(seq1 within seq2) // is equivalent to:
((1[*0:$] ##1 seq1 ##1 1[*0:$]) intersect seq2 )
```

The '**within'** sequence operator `(seq1 within seq2)` just specifies that `seq1` must exist within the range of `seq2`. If `seq1` is one cookie and seq2 is a jar, then `(seq1 within seq2)` asserts that there is at least one occurrence of a cookie in the jar; but there could also be **more** than one cookie! Thus, the English understanding of the '**within'** operator can be misleading.

In my paper: *Understanding SVA Degeneracy*[iii], I provide a case study involving the '**within'** operator to illustrate potential issues that may arise when it is used with an empty match. The paper addresses an actual user case where the requirements stated that "between two rises of `req`, and starting from the next cycle, there should be one and only one new `gnt`". Intuitively, or perhaps naively, one would write something like the following property:

```
$rose(req) |=> $rose(gnt)[=1] within $rose(req)[->1]
```

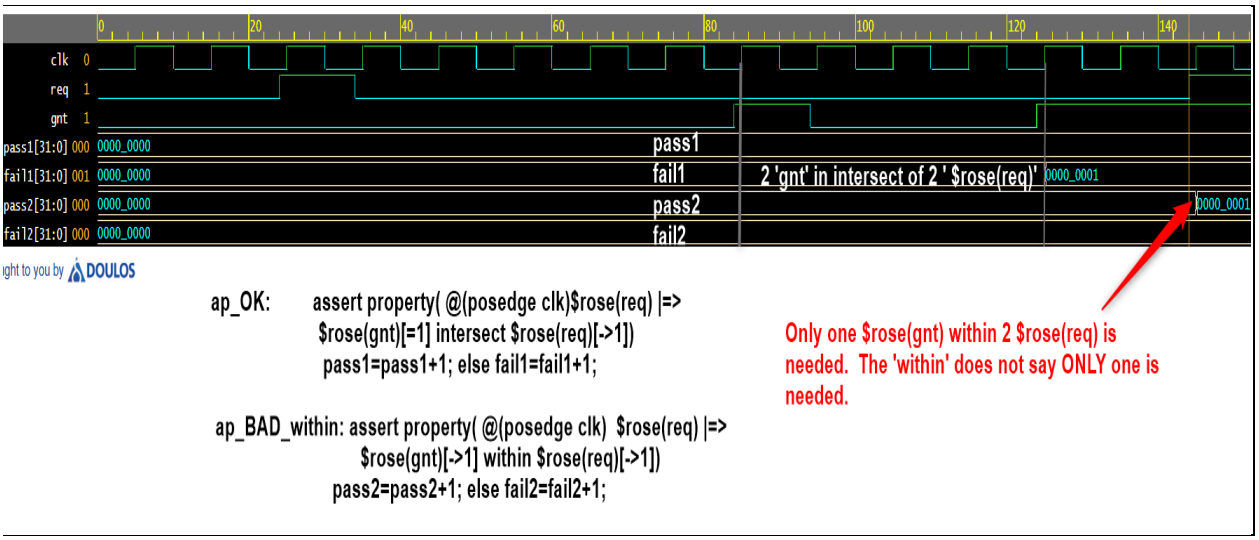
believing that we guarantee only 1 occurrence of `$rose(gnt)` because of the **`[=1]`** operator

`b[=1]` is equivalent to: `!b[*0:$] ##1 b ##1 !b[*0:$]`

In the referenced paper, the simulation of the above property showed that it failed to throw an assertion error when 2 or more new occurrences of `gnt` occurred. Below is a simulation of the same model but with the goto operator instead of the non-consecutive repetition as it too has the same issue.

```
$rose(gnt)[->1] within $rose(req)[->1])
```

https://www.edaplayground.com/x/cWAH code



6.0 Efficiency

In his presentation on SystemVerilog Assertions for Formal Verification[iv], Dmitry Korchemny highlighted an important point regarding sequence operators from the intersection family (such as **`intersect, and, within, throughout,`** and **`until`** (*though it is a property where the RHS and LHS properties are expressions or sequences*). He noted that these operators can be computationally expensive and potentially inefficient in formal verification, although they may not pose significant performance issues in simulation environments.

As a general guideline for writing effective assertions, we (*Ajeetha Kumari, and I*) recommend using multiple smaller and simpler properties rather than complex ones that rely heavily on the intersection family of operators. This approach offers several benefits:

1. It improves the readability and maintainability of the assertions.
2. It often leads to better performance in formal verification tools.
3. It makes debugging easier, as simpler assertions are typically more straightforward to analyze when they fail.
4. It allows for more granular analysis of design behavior, potentially uncovering issues that might be obscured in more complex, monolithic assertions.

By following this guideline, verification engineers can create more efficient and effective assertion sets that are well-suited for both simulation and formal verification methodologies.

7.0 Summary and Conclusions

The **'until, until_with'** operators share a similar challenge with the **'throughout'** operator in that it involves an implicit repetition that must be mentally visualized and understood by the user. This implicit nature can lead to misinterpretations if not carefully considered. Furthermore, the **'until'** operator presents an additional complexity due to its variant form, **'until_with'**.

The '**within**' operator can be a source of errors if its true meaning is not fully comprehended by users. This potential for misunderstanding is particularly pronounced when the left-hand side (LHS) sequence is terminated with an empty match option. The complexity arises from two main factors:

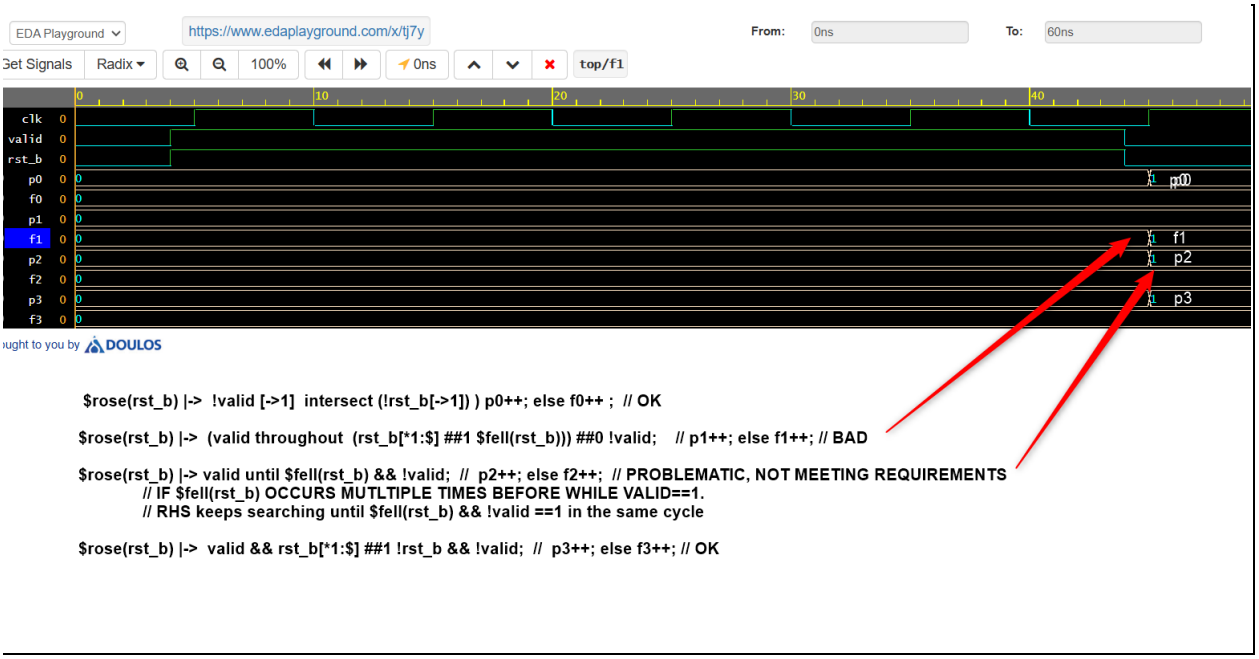
1. Interpretation of the operator: The natural language meaning of **'within'** might lead users to make incorrect assumptions about its behavior in SVA.
2. Empty match scenarios: When the LHS sequence can terminate with an empty match, it introduces subtle behaviors that may not be immediately obvious.

When sequence operators from the intersection family are needed, I advocate for the use of the **'intersect'** operator over its higher-level counterparts. The **'intersect'** operator offers greater precision and clarity, leaving less room for ambiguity. Providing a more explicit representation of the intended behavior significantly reduces the risk of misinterpretations that often stem from seemingly intuitive, but potentially misleading, natural language-like constructs. This approach promotes more accurate and reliable assertion writing in SVA, ultimately leading to more robust verification processes. Keep in mind though that efficiency needs to be considered when writing for formal verification.

The simulation of model addressed in this paper is available at:

https://www.edaplayground.com/x/tj7y

The simulation results are shown below.



Acknowledgments:
I thank Ajeetha Kumari for providing valuable comments about this topic. Ajeetha is a co-authors of my book *SystemVerilog Assertions Handbook Revised 4th Edition 2023*. Ajeetha is currently driving SystemVerilog testbench linting at AsFigo.
https://asfigo.com/
https://www.linkedin.com/in/ajeetha/

I also thank Ed Cerny providing feedback (https://www.linkedin.com/in/eduard-cerny-a40901/).
Ed is a co-author of the book *SVA: The Power of Assertions in SystemVerilog 2nd Edition* and was a member of the IEEE committee, representing Synopsys, for the definition of SystemVerilog Assertions.

I thank Doulos for the access to https://www.edaplayground.com/

I thank Siemens for granting me a license of Questa™ simulator, as that allowed me to dwell deeper into the threads.

I effectively used Screenpresso to capture and annotate the simulation results.
https://www.screenpresso.com/

I used Perplexity AI to assist with idea expansion and English writing. This approach effectively combines AI-assisted research efficiency with human insight and domain expertise.

Link to the list of papers and books that I wrote, many are now donated.
https://systemverilog.us/vf/Cohen_Links_to_papers_books.pdf

[i] Reflections on Users' Experiences with SVA, part 2
https://verificationacademy.com/verification-horizons/july-2022-volume-18-issue-2/reflections-on-users-experiences-with-sva-part-2
Addresses the usage of these four relationship operators: **`throughout, until, intersect, implies`**

[ii] Question originated from a post
https://verificationacademy.com/forums/t/sva-throught-operator/48893
https://www.linkedin.com/feed/update/urn:li:activity:7261863740399058944/

[iiiiii] Paper: Understanding SVA Degeneracy
https://systemverilog.us/vf/Degeneracy111723Ben.pdf

[iv] SystemVerilog Assertions For Formal Verification
Dmitry Korchemny, Intel Corp.
HVC2013, November 4, 2013, Haifa
https://bit.ly/3ObJBXO