

If Chained Implications in Properties Weren't So Hard, They'd be Easy

Don Mills

don.mills@microchip.com

mills@lcdm-eng.com

Microchip Technology Inc.
Chandler, AZ, USA

www.microchip.com

ABSTRACT

The use of implication operators in concurrent assertion properties is critical to masking false negatives during verification. However, many designers shy away from using multiple implications within the same property because they are difficult to understand and maintain. This paper will dissect a two-level chained implication in which the first level consequent second level antecedent contains an eventuality condition (an unbounded range). Analysis will consider numerous scenarios and what-if's regarding why and how this property works the way it does.

Table of Contents

INTRODUCTION.....	4
WHAT AND WHY OF SYSTEMVERILOG ASSERTIONS.....	4
IMMEDIATE ASSERTION.....	4
CONCURRENT ASSERTIONS	5
<i>Property and Sequence Blocks</i>	6
<i>Implication Operator</i>	7
<i>Range Repetition vs. Cycle Delay Range</i>	11
ANALYSIS OF A COMPLEX PROPERTY	13
SIMPLE TEST TO THE COMPLEX PROPERTY	15
CORRECT WAY TO MODEL THE COMPLEX PROPERTY USING <code>FIRST_MATCH</code>	16
ALTERNATE WAY OF MODELING THE COMPLEX PROPERTY USING <code>##0</code>	17
THE SECOND IMPLICATION REPLACED BY <code>##0</code>	18
TWO PROPERTIES	19
USE AN UPPER RANGE BOUND.....	20
ALLOW ONLY THE FIRST ANTECEDENT TO BE VACUOUS.....	22
CONCLUSIONS	23
REFERENCES.....	24
ABOUT THE AUTHOR	24

Table of Figures

Figure 1 – Sequence and Property Pass/Fail for the Thread Starting at cycle 1 Inclusive Example 4.	7
Figure 2 – Sequence and Property Pass/Fail for all the Threads Between Cycle 0 and Cycle 10 for Example 4.	8
Figure 3 – Property Pass/Fail for all the Threads Between Cycle 0 and Cycle 10 for Example 6. Vacuous Successes are not Noted.	9
Figure 4 – Sequence <code>bus_req</code> and Property <code>bus_req_prop7</code> Pass/Fail Results for Example 7	10
Figure 5 – Sequence <code>bus_req</code> and Property <code>bus_req_prop7</code> Pass/Fail Results for Example 7	10
Figure 6 – Sequence and Property Pass/Fail results for Example 8.....	11
Figure 7 – Sequence and Property Pass/Fail Results for Example 9.	12
Figure 8 – Sequence and Property Pass/Fail Results for Example 9.	13
Figure 9 – Chained Implication Labels.....	14
Figure 10 – Failing Test for Chained Implications.....	15
Figure 11 – Passing Test? For Chained Implications.....	15
Figure 12 – Neither Passing or Failing Test for Chained Implications.	16
Figure 13 – Real Passing Test for Chained Implications Using <code>first_match</code>	17

Figure 14 – Fusion Operator Replacing the First Implication Operator	18
Figure 15 – Fusion Operator Replaces Second Implication Operator.....	18
Figure 16 – Fusion Operator Replaces the Second Implication Operator	19
Figure 17 – Bounded Range With and Without a first_match.....	20
Figure 18 – Variable Upper Range Bound	22

List of Examples

Example 1 – Self Contained Concurrent Assertion Directive	5
Example 2 – Concurrent Assertion Calling a Property Block.....	6
Example 3 – Assertion, Property and Sequence Blocks Used Together	6
Example 4 – Sequence with Multiple Endpoints True	7
Example 5 – Property Showing the Antecedent and Consequent of an Implication.....	8
Example 6 – Range in Consequent – Automatic First-Match is Applied	9
Example 7 – Sequence with Multiple Endpoints used in Antecedent	9
Example 8 – Antecedent Sequence with first_match Applied to Possible Multiple Endpoints	11
Example 9 – Sequence with Multiple Endpoints in Antecedent	12
Example 10 – Clock Cycle Counter and Max Count Comparator	14
Example 11 – Chained implication with a unbounded repetition range in an antecedent ..	14
Example 12 – Chained Implication Using first_match on an Unbounded Repetition Range in Antecedent.....	16
Example 13 – Replace First Implication With Fusion.....	17
Example 14 – Replace Second Implication with Fusion.....	18
Example 15 – Chained Implication with first_match vs. Fusion Replacing the Second Implication.....	19
Example 16 – Chained Implication with Bounded Repetition Range in Antecedent	20
Example 17 – Variable Used to Set the Upper Range Bound	21
Example 18 – Variable Used to Set Upper Range Bound, Only Test When Upper Bound is Reached.....	22
Example 19 – Chained Implication Where Only First Antecedent Can be Vacuously True	23

1 Introduction

Assertions have become a key component in both the design and verification phases of a project. Liberal use and accurate modelling of assertions, which includes properties and sequences, provide continuous visibility to a design during verification. In addition to visibility, concurrent assertions are very useful for monitoring interaction and protocol between signals in the design.

Another feature of assertions is that they can very precisely describe the intent of a design. Assertions can be used as a “second source” for design verification since the nomenclature used to describe assertions is significantly different from the standard RTL/synthesizable modelling. Finally, with a significant scattering of assertions throughout a design, a formal tool can make use of these assertions to analyze and compare the design described by the assertions to the RTL model of the design.

The syntax and modelling for assertions can be considered very cryptic. This is, of course, the reason why assertions are so concise. However, with this cryptic, concise syntax comes a learning curve that can be difficult to climb. Then once understood, the syntax is difficult to maintain or remember if it is not used regularly.

This paper will begin by giving a brief definition of the key components of assertions and some basic uses for assertions. The paper does expect that the audience has a basic understanding of SystemVerilog Assertions. A more complex model will then be introduced and this model will be the basis for the remainder of the paper. Variations of this complex model will be discussed, analyzed, and compared to the original, as a means of understanding the details of how the original model works.

2 What and Why of SystemVerilog Assertions

SystemVerilog has two types of assertions: *immediate assertions* and *concurrent assertions*. Both types of assertions are used to perform tests on a design whenever the assertion is called or executed. When an assertion test is completed, a pass or fail statement from the assertion can be executed. Assertions provide a mechanism for continuous monitoring of signals and conditions across all simulations.

2.1 Immediate Assertion

Immediate assertions execute in zero simulation time, i.e., they execute immediately. An immediate assertion is a procedural statement, and therefore can be placed anywhere a procedural statement can be placed: within always blocks, initial blocks, tasks, and functions. Immediate assertions are very similar to if/else statements in that they can have both pass and fail procedural statements. However they differ from if/else statements in many ways including:

1. Assertions are ignored by synthesis.

2. Assertion execution can be disabled and enabled during simulation.
3. Assertions are intended for monitoring a design rather than modeling a design.
4. Assertions execute severity level tasks if the assertion fails.

The syntax for an immediate assertion is:

```
assert (expression) [pass_statement;] [else fail_statement;]
```

Note that the `pass_statement` and the `else fail_statement` are optional. If the `fail_statement` is left off, the default severity level of `$error` will execute.

For examples and details on using immediate assertions to trap logic X and Z problems, refer to the paper “*Being Assertive With Your X*” [4], published in the proceedings of SNUG 2004.

2.2 Concurrent Assertions

Concurrent assertions use a clock or some other repetitive signal (referred to hereafter as the property clock) to trigger the assertion evaluation. The primary difference between immediate and concurrent assertions is that concurrent assertions evaluate conditions over time, whereas immediate assertions test at the point in time when the assertion is called. The syntax difference between the two types of assertions is very slight. The concurrent assertion directive includes the key word `property`, whereas the immediate assertion does not. The syntax for a concurrent assertion directive is:

```
assert property (property_expr) [pass_statement;] [else fail_statement;]
```

The argument to `assert property` is a *property expression* which differs from the argument of an immediate assertion, which is a simple Boolean expression. A property expression is comprised of a clock specification and a sequence of Boolean expressions tested over time. The expressions are evaluated on the clock edge specified. The sequence of Boolean expressions can be spread over multiple clock cycles by using the `##` cycle delay operator between each expression.

The following code is an example of a completely self-contained concurrent assertion directive.

```
example_1:assert property
  (@(posedge clk) ( req ##1 grant ##10 !req ##1 !grant))
  else $error("bus request failed");
```

Example 1 – Self Contained Concurrent Assertion Directive

The sequence in the example above is read as: “req should be true (high) immediately, followed by grant being true (high) one clock cycle later. After ten more clock cycles, req should be false (low), followed by grant being false (low) one clock cycle later.” For this assertion to succeed, each expression must evaluate true at its specified time.

2.2.1 Property and Sequence Blocks

The property expression of a concurrent assertion can be defined in a separate block of code, between the keywords `property` and `endproperty`. This enables the same property expression to be re-used by multiple concurrent assertions.

```
property bus_req_prop2;  
    @(posedge clk) req ##1 grant ##10 !req ##1 !grant;  
endproperty:bus_req_prop2  
  
example_2:assert property (bus_req_prop2)  
    else $error("bus request failed");
```

Example 2 – Concurrent Assertion Calling a Property Block

A complex property expression can be broken into smaller sequence building blocks, specified between `sequence` and `endsequence`. This is illustrated in the following example.

```
sequence start_bus_req;  
    req ##1 grant;  
endsequence:start_bus_req  
  
sequence end_bus_req;  
    !req ##1 !grant;  
endsequence:end_bus_req  
  
property bus_req_prop3;  
    @(posedge clk) start_bus_req ##10 end_bus_req;  
endproperty:bus_req_prop3  
  
example_3: assert property (bus_req_prop3);
```

Example 3 – Assertion, Property and Sequence Blocks Used Together

One difference between a property and a sequence is that property expressions contain an implicit first-match whereas a sequence does not. This means that if a sequence has multiple pass conditions, then each will occur. However, when that same sequence is placed in a property expression, only the first pass condition of the sequence will be observed by the property, ending that thread. Code Example 4 and Figure 1 below illustrate this concept. **sequence** `bus_req` in Figure 1 shows four passes per the signal relationship of `req` and `grant` whereas **property** `bus_req_prop4` will only see the first successful pass of **sequence** `bus_req`, causing that thread of the property to exit. Note that Figure 1 only shows the pass/fail relationship for the sequence and the property for the specific thread starting at cycle 1. The significance of this concept will be shown later in this paper as part of the discussion of implication operators.

```

sequence bus_req;
  req ##[1:5] grant; // equivalent to:
                    // req ##1 grant or
                    // req ##2 grant or
                    // req ##3 grant or
                    // req ##4 grant or
                    // req ##5 grant

endsequence:bus_req

property bus_req_prop4;
  @(posedge clk) bus_req;
endproperty:bus_req_prop4

example_4: assert property (bus_req_prop4);

```

Example 4 – Sequence with Multiple Endpoints True

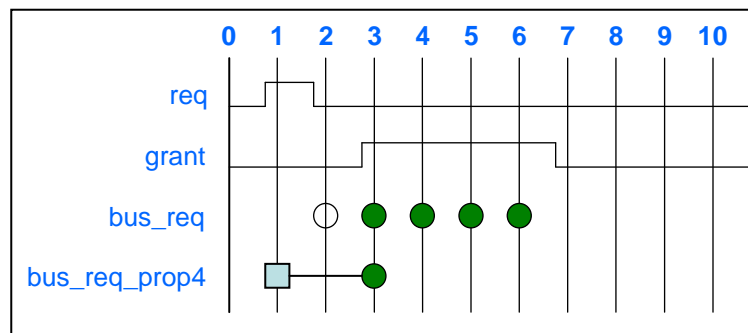





Figure 1 – Sequence and Property Pass/Fail for the Thread Starting at cycle 1 Inclusive Example 4.

Note: For waveform figures, a filled square will denote the beginning of a thread. Bubbles will denote thread endpoints. A clear bubble will denote a property or sequence failure and a filled bubble will denote a property or sequence pass.

	Start of thread		pass
			fail

2.2.2 Implication Operator

Most concurrent assertions are written so that the assertion “*fires*” each and every clock cycle, throughout simulation. This allows the assertion to run in the background, concurrent with the design functionality. Since the assertion fires every clock cycle, an assertion with a sequence that takes twelve clock cycles to execute could possibly have twelve concurrent threads running at the same time, with each thread starting on each subsequent clock cycle. In the bus request/grant sequence examples above, req will be tested every clock cycle, starting a new concurrent assertion thread. If req is true, the thread will continue and test for grant on the

next clock cycle. If `req` is false, however, the assertion will fail at that point in time. This would be a false failure, since the assertion is testing for a sequence that starts with `req` testing true. Figure 2 below expands the pass/fail conditions for `bus_req_prop4` showing all the pass/fails for each thread, where Figure 1 only shows the results for the thread starting at cycle 1.

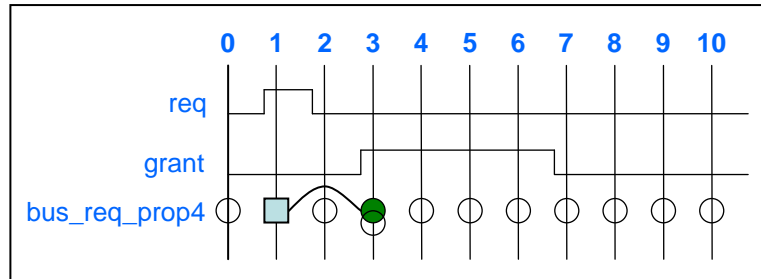


Figure 2 – Sequence and Property Pass/Fail for all the Threads Between Cycle 0 and Cycle 10 for Example 4.

In Figure 2, cycle 1 has neither a pass nor a fail because there are no threads ending at that point in time. Cycle 3 has both a pass and a fail due to two separate threads ending at that cycle. The thread that started at cycle 1 successfully completed at cycle 3 with a pass. The thread that started at cycle 3 ended immediately with a fail, because `req` is false at that cycle.

The behavior of the assertion modeled in Example 4 and fully shown in Figure 2 is not really practical due to an assertion failure occurring almost every cycle. To make assertions usable, the assertion property needs to be modeled so that it will only test during expected event cycles and be idle during don't-care cycles. SystemVerilog properties make this possible by using the implication operator. Typically, assertions property expressions are specified with an implication operator, either overlapping `|->` or non-overlapping `!=>`. An implication operator tells the property not to evaluate a property expression (the consequent) following the operator unless the first condition before the operator (the antecedent) is true.

```
property example_5;
  @(posedge clk) antecedent_sequence_expression |->
    consequent_property_expression;
endproperty:example_5
```

Example 5 – Property Showing the Antecedent and Consequent of an Implication

In the request/grant code examples previously discussed, the designer will most likely only want to test the request/grant sequence when `req` is true. For clock cycles where `req` is false, the assertion is a don't-care, and the request/grant sequence should not be evaluated. In assertion terms, this condition is called a vacuous success. In the following example, the implication operator prevents (guards) the consequent expression from testing when `req` is not true. The assertion does not fail; it simply does not run and returns a vacuous success.


```

property bus_req_prop6;
  @(posedge clk) req |-> ##[1:5] grant;
endproperty:bus_req_prop6

example_6: assert property (bus_req_prop6);

```

Example 6 – Range in Consequent – Automatic First-Match is Applied

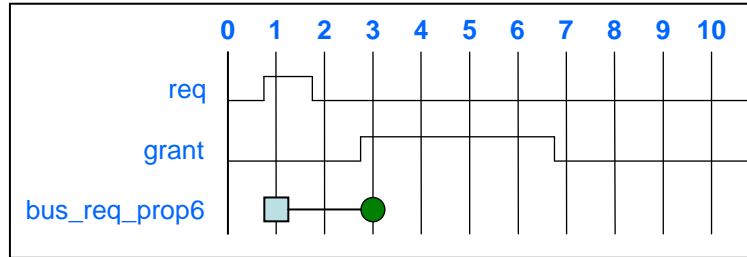


Figure 3 – Property Pass/Fail for all the Threads Between Cycle 0 and Cycle 10 for Example 6. Vacuous Successes are not Noted.

Example 6 and Figure 3 show the request/grant code from Example 4, but with an implication operator guarding against testing when `req` is low. The sequence from Example 4 is partitioned into two parts: the antecedent or cause, and the consequent or effect. Note that the implication operator is a property expression operator and cannot be used in a sequence.

The above example illustrates that the implicit first-match holds true for the consequent of an implication that contains a sequence with a range. However, when the antecedent contains a sequence with a range, each passing condition of the range starts a unique thread. This unique thread in turn, starts a consequent evaluation. For the overall property expression to match, each passing antecedent must have a matching consequent.

```

sequence bus_req;
  req ##[1:5] grant;
endsequence:bus_req

property bus_req_prop7;
  @(posedge clk) bus_req |-> ##[1:5] done;
endproperty:bus_req_prop7

example_7: assert property (bus_req_prop7);

```

Example 7 – Sequence with Multiple Endpoints used in Antecedent

The code from Example 7 is the basis for the wave diagrams in Figures 4 and 5 below. `bus_req` in each figure indicates the pass/fail for the sequence used in the property antecedent. Because this sequence has a range with multiple passing conditions, each passing condition must have a passing consequent in order for the overall property expression to pass. In Figure 4, signal `done` is true at cycle 5, which provides a pass condition for the sequence threads starting at cycle 3 and 4. However sequence threads starting at cycle 5 and 6 do not have any

corresponding passing consequent and therefore the property fails. The implicit property first-match does not apply in this case, because all the passing conditions in the antecedent are part of the singular property expression inclusive. Each passing condition from cycle 2 through cycle 6 must match to a passing consequent for the property to pass.

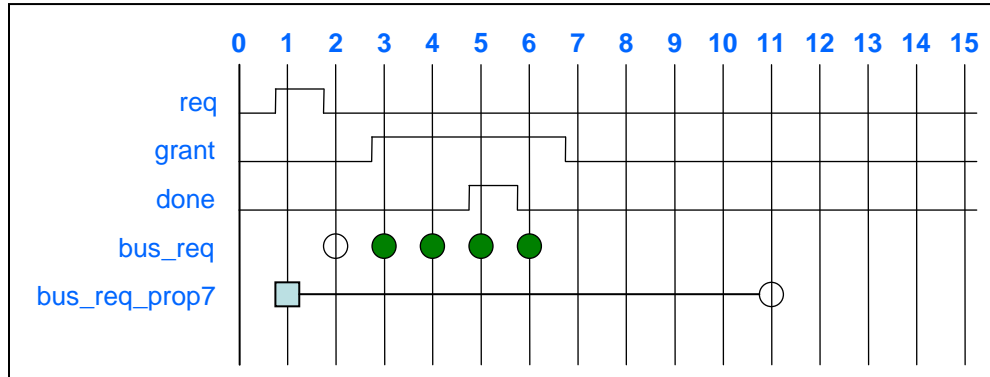


Figure 4 – Sequence bus_req and Property bus_req_prop7 Pass/Fail Results for Example 7

The signals in Figure 5 below follow the same scenario as the signals in Figure 4, with the exception that a done occurs at cycle 9. This provides a passing condition for the sequence threads that started at cycle 5 and 6.

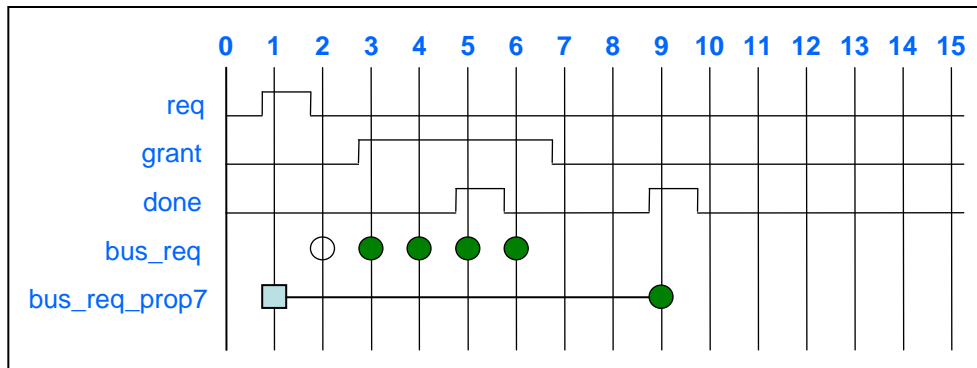


Figure 5 – Sequence bus_req and Property bus_req_prop7 Pass/Fail Results for Example 7

When using a sequence that has the possibility of multiple passing conditions in an antecedent, a much more practical model is to directly apply the **first_match** operator to the sequence. Then, at most only one passing condition from the antecedent will be considered for the property expression to pass. Example 8 and Figure 6 below show how the same inputs used in Figure 5 provide a passing property expression based on the first passing antecedent condition.

```

sequence bus_req;
  req ##[1:5] grant;
endsequence:bus_req

property bus_req_prop8;
  @(posedge clk) first_match(bus_req) |-> ##[1:5] done;
endproperty:bus_req_prop8

example_8: assert property (bus_req_prop8);

```

Example 8 – Antecedent Sequence with `first_match` Applied to Possible Multiple Endpoints

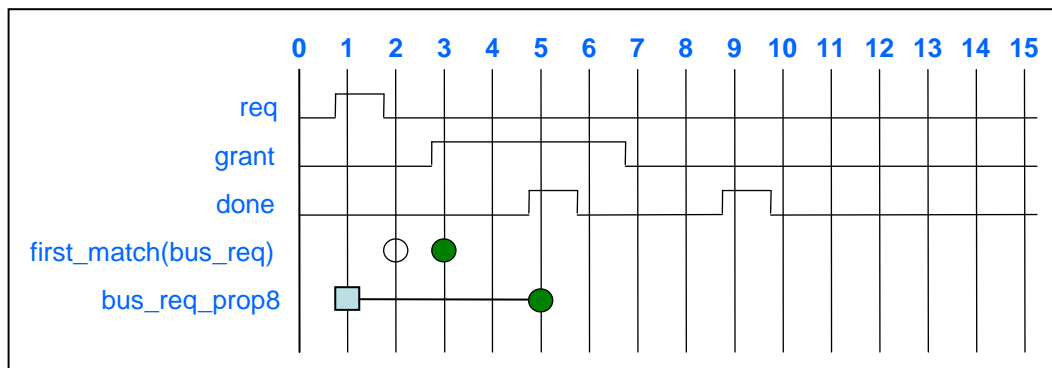


Figure 6 – Sequence and Property Pass/Fail results for Example 8.

2.2.3 Range Repetition vs. Cycle Delay Range

In the examples discussed to this point, the sequences used a cycle delay range. SystemVerilog assertions have other types of ranges that are associated with the repetition of Boolean or sequential expressions. The point to note here is that as long as the range repetition is not failing, the range rules previously discussed regarding antecedents apply to both range repetition and a cycle delay range. For example, in the code below, as long as `req` is high within the range, the sequence expression will look for a `grant` to occur. Following the example are two figures that illustrate the consecutive range repetition.

```

sequence bus_req;
  req[*1:5] ##1 grant;
endsequence:bus_req

property bus_req_prop9;
  @(posedge clk) bus_req |-> ##[1:2] done;
endproperty:bus_req_prop9

property bus_req_prop_fm9;
  @(posedge clk) first_match(bus_req) |-> ##[1:2] done;
endproperty:bus_req_prop_fm9

```

```

example_9:  assert property (bus_req_prop9);
example_fm9: assert property (bus_req_prop_fm9);

```

Example 9 – Sequence with Multiple Endpoints in Antecedent

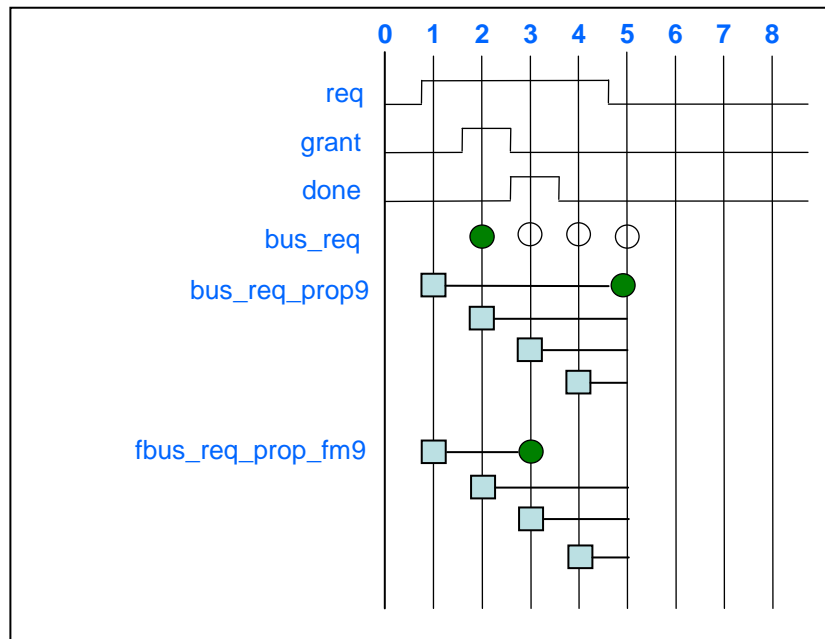


Figure 7 – Sequence and Property Pass/Fail Results for Example 9.

In Figure 7, `bus_req_prop9` has a thread with a passing condition at cycle 3, but does not show the thread passing until cycle 5. Remember, all passing conditions of the antecedent for a multiple match sequence must each have a passing consequent. For `bus_req_prop9`, the property expression had to wait until `req` went away to ensure that all the possible passing antecedents could be tested. Thus the test for the first thread shows passing at cycle 5. The remaining threads starting with `req` high at cycles 2, 3, and 4, never started an actual antecedent pass.

The contrast between `bus_req_prop9` and `bus_req_prop_fm9` should be obvious. The **first_match** operator will not wait for all the passing antecedents to show matching consequents. Rather, **first_match** makes each passing antecedent a separate and unique thread. Each time an antecedent passes, it will test for a passing consequent, and then the property expression is completed for that thread. This concept is easier to visualize, as shown in Figure 8.

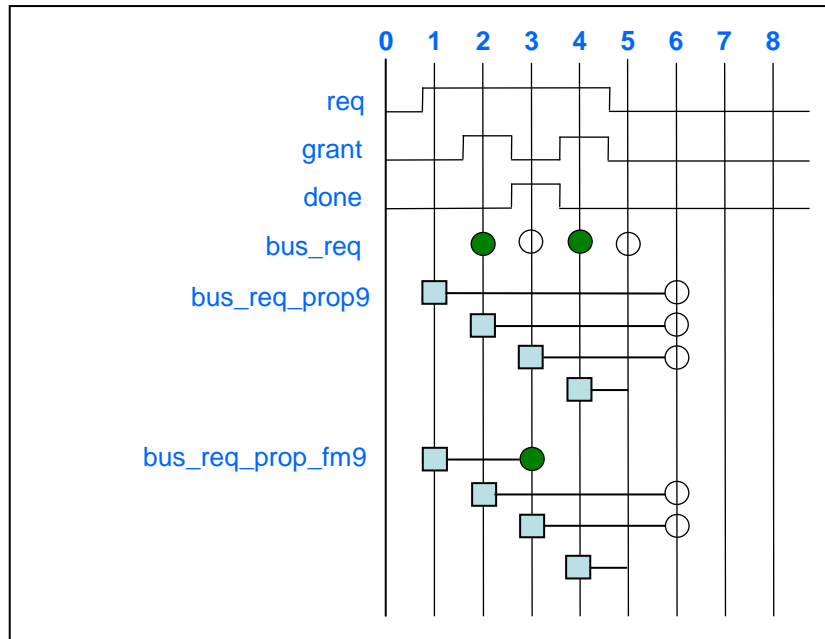


Figure 8 – Sequence and Property Pass/Fail Results for Example 9.

In Figure 8, `bus_req_prop9` has a thread with a passing condition at cycle 3, and a failing condition at cycle 6. Combined, this means that the property expression fails due to all the passing antecedents not having passing consequents.

In contrast between `bus_req_prop9` and `bus_req_prop_fm9`, the **first_match** will isolate the first passing antecedent from the rest of the possible sequence matches in the antecedent. Thus, once a passing condition occurred at cycle 3, the property expression passed. Other threads are started at other cycles and they either fail or just end, as shown in the figure.

The concept that a range in the antecedent holds the property expression from completing is the basis for the complex property discussed in the remainder of the paper. The complexity of this property is magnified by the fact that the range is modeled as an unbounded range. As shown in Figure 7 and Figure 8, this means that as long as the antecedent has a passing condition or the possibility of a passing condition in the future, the property expression will never end.

3 Analysis of a Complex Property

The background of assertions and property expressions discussed in Section 2 is provided to give a foundation for discussion of the property expression introduced in this section. Suppose a designer wanted to monitor the number of cycles between a start pulse and an end pulse and then verify some condition based on the number of cycles that occurred. For example, in a burst memory write, an assertion could be used to ensure that the final memory address matches the initial memory address plus the number of clocks (assuming a write occurred on each clock). Another scenario might be to verify that the number of clocks between the start and end

conditions is less than some maximum value. Such tests could be verified with a property using chained implication operators as shown in Example 10.

```

`define TRUE 1

property prop_10;
  int v_cnt;
  @(posedge clk) ($rose(start), v_cnt = 0) |->
    (`TRUE, v_cnt++)[*0:$] ##1 done |->
      (v_cnt <= MAX);
endproperty:prop_10

example_10: assert property (prop_10);

```

Example 10 – Clock Cycle Counter and Max Count Comparator

The `TRUE is needed so that v_cnt would increment on every clock until done becomes true. For the property expression to increment on each clock cycle, the consecutive range repetition is needed. If a cycle delay range ##[0:\$] were used instead of the [*0:\$], the execution of v_cnt++ would occur only once.

Chaining implication operators adds a level of complexity to a property expression that engineers often prefer to avoid. The complexity in Example 10 is then magnified because of the unbounded repetition in what appears to be both a consequent and an antecedent.

In order to simplify the discussion for this paper, a simpler version of the property from Example 10 will be used as shown in Example 11 below.

```

property prop_11;
  @(posedge clk) $rose(a) |-> b[*0:$] ##1 c |-> d;
endproperty:prop_11

example_11: assert property (prop_11);

```

Example 11 – Chained implication with an unbounded repetition range in an antecedent

The property expression in Example 11 can be labeled as shown in the following figure.

$\color{red}{\$rose(a)} \quad | \rightarrow \quad \color{blue}{b[*0:\$]} \quad \color{red}{##1} \quad \color{blue}{c} \quad | \rightarrow \quad \color{blue}{d};$
 $\color{red}{\underbrace{\hspace{1.5cm}}_{A1}} \quad \color{blue}{\underbrace{\hspace{1.5cm}}_{a2}} \quad \color{red}{\underbrace{\hspace{1.5cm}}_{C1}} \quad \color{blue}{\underbrace{\hspace{1.5cm}}_{c2}}$

Figure 9 – Chained Implication Labels

Figure 9 shows the label for the first antecedent as **A1**, and everything to the right of the first implication operator as consequent **C1**. The code between the two implication operators is

labelled as **a2** and is the antecedent to the second implication operator. Finally the consequent to the right of the second implication operator is labelled **c2**. These labels are referred to frequently throughout the remainder of this paper.

3.1 Simple Test to the Complex Property

Let us begin to look at this property by applying a simple test to it. To keep consistency between code Example 10 and code Example 11, condition b will remain true through most of the tests that follow.

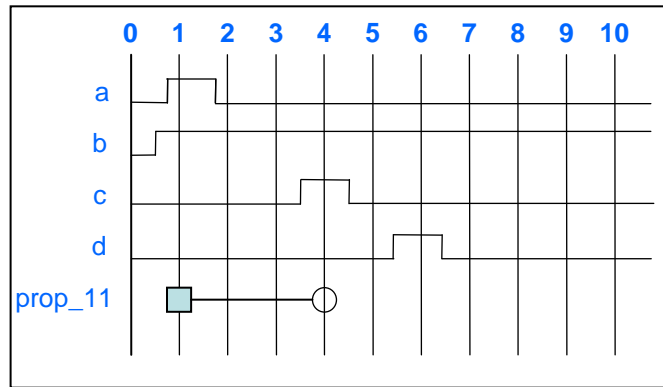


Figure 10 – Failing Test for Chained Implications.

Property expression `prop_11` will fail when conditions `c` and `d` are both not true on the same cycle after the property expression has started. Figure 10 shows the property stopping at cycle 4 with a failure.

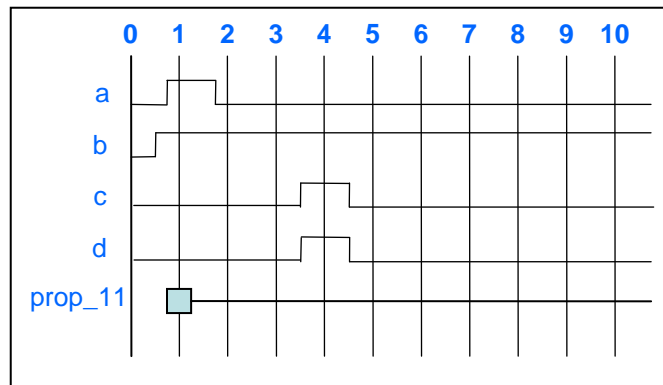


Figure 11 – Passing Test? For Chained Implications.

Why does the property in Figure 11 not stop at cycle 4 with a pass? Is there not an implied first-match applied to consequents? There is, but in this case, the consequent has not had a complete passing match yet. Referring to the code diagram above and Figure 9, the condition in **a2** has the repetition range applied to condition `b`. For the property expression **a2/c2** to pass, all

passing conditions of **a2** must have a matching passing condition of **c2**. With **b** repeating true, the test will continue to look for matching **c**'s and **d**'s. As long as **b** is true and each occurrence of **c** and **d** match, the **a2/c2** expression does not end, and therefore does not pass.

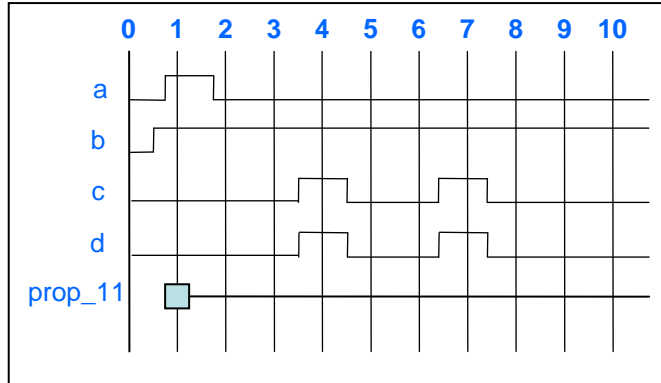


Figure 12 – Neither Passing or Failing Test for Chained Implications.

In summary, as long as **b** is true, the property expression **a2/c2** will test forever looking for the condition of matching **c**'s and **d**'s.

3.2 Correct Way to Model the Complex Property Using `first_match`

In order to model the code from code Example 11, the `first_match` operator must be applied to antecedent **a2** (Figure 9). Then, only the first matching condition of **a2** (Figure 9) will be tested for a matching consequent, therefore limiting the unbounded range of **b**.

```
property prop_12;
  int v_cnt;
  @(posedge clk) $rose(a) |-> first_match(b[*0:$] ##1 c) |-> d;
endproperty:prop_12

example_12: assert property (prop_12);
```

Example 12 – Chained Implication Using `first_match` on an Unbounded Repetition Range in Antecedent

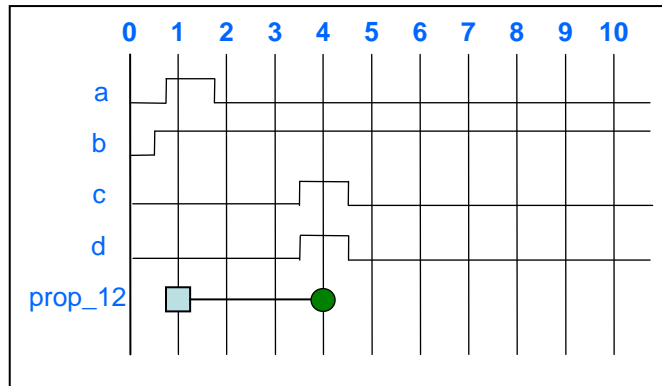


Figure 13 – Real Passing Test for Chained Implications Using `first_match`

3.3 Alternate way of modeling the complex property using `##0`

An alternate way of modelling a chained implication property expression is to replace the first implication operator with the fusion operator. The fusion operator `##0` represents an overlapping concatenation in sequences, where that the last sub-element of the first sequence and the first sub-element of the second sequence hold true in the same cycle. This is another way of stating that the second sequence starts at the same cycle in which the first sequence completes.

```

property prop_13a;
  int v_cnt;
  @(posedge clk) $rose(a) ##0 (b[*0:$] ##1 c) |-> d;
endproperty:prop_13a

property prop_13b;
  int v_cnt;
  @(posedge clk) $rose(a) ##0 first_match(b[*0:$] ##1 c) |-> d;
endproperty:prop_13b

example_13a: assert property (prop_13a);
example_13b: assert property (prop_13b);

```

Example 13 – Replace First Implication With Fusion

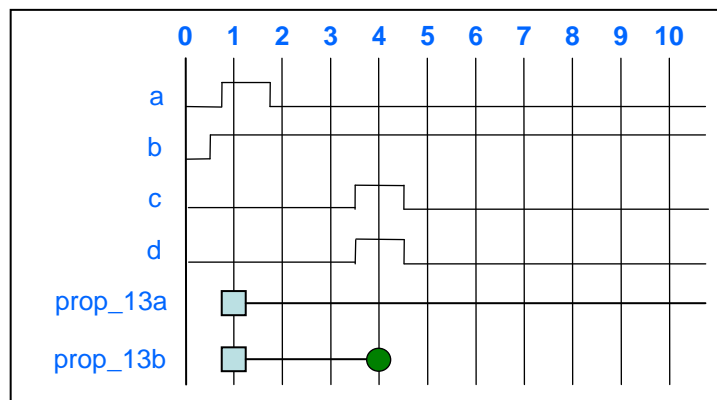


Figure 14 – Fusion Operator Replacing the First Implication Operator

Replacing the first implication operator with the fusion operator may make the overall property expression easier to understand, and it does not change the functionality. Essentially, `prop_13a` and `prop_13b` are functionally equivalent to `prop_11` and `prop_12`, respectively.

3.4 The Second Implication Replaced by ##0

If an alternate way of modelling a chained implication property expression is to replace the first implication operator with the fusion operator, then could the second implication operator be replaced with a fusion operator?

```

property prop_14a;
    int v_cnt;
    @(posedge clk) $rose(a) |-> b[*0:$] ##1 c ##0 d;
endproperty:prop_14a

property prop_14b;
    int v_cnt;
    @(posedge clk) $rose(a) |-> first_match(b[*0:$] ##1 c ##0 d);
endproperty:prop_14b

example_14a: assert property (prop_14a);
example_14b: assert property (prop_14b);
    
```

Example 14 – Replace Second Implication with Fusion

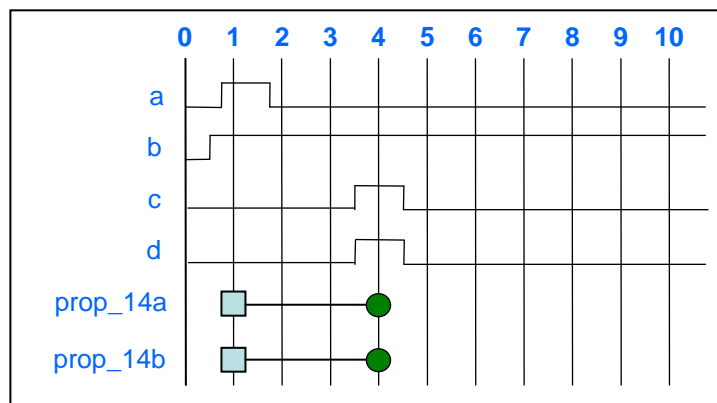


Figure 15 – Fusion Operator Replaces Second Implication Operator

On first glance, one could conclude that replacing the second implication operator with a fusion operator gives the desired implementation with or without the `first_match` operator. However, there are corner conditions that do not match. For instance, a vacuous success for `a2` (Figure 9) will return a vacuous success for the entire property expression containing the chained

implications. These same conditions that caused a vacuous success for the chained model will cause the fusion model to fail.

```

property prop_15a;
  int v_cnt;
  @(posedge clk) $rose(a) |-> first_match(b[*0:$] ##1 c) |-> d;
endproperty:prop_15a

property prop_15b;
  int v_cnt;
  @(posedge clk) $rose(a) |-> b[*0:$] ##1 c ##0 d;
endproperty:prop_15b

example_15a: assert property (prop_15a);
example_15b: assert property (prop_15b);

```

Example 15 – Chained Implication with `first_match` vs. Fusion Replacing the Second Implication

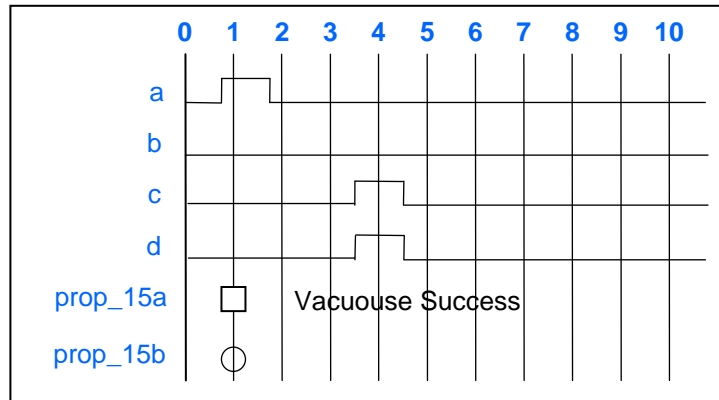


Figure 16 – Fusion Operator Replaces the Second Implication Operator

Property expression `prop_15a` has the chained implication operators and will return a vacuous success at cycle 1 because antecedent `a2` (Figure 9) is false. Property expression `prop_15b` fails at cycle 1 because it uses the fusion operator instead of a second implication.

3.5 Two Properties

An approach used by designers to avoid chaining implications is to use multiple property blocks, each with a single implication. For instance, a designer would replace the property expression `a |-> b |-> c` with two separate property expressions `a |-> b` and `b |-> c`, each in its own property block. These two separate properties do not represent the same models as the single property expression. The separate property expression with `b` in the antecedent will test every cycle, and is different from the chained property expression where `b` will only be tested when `a` passes. If the designer does not want to not use implication chaining, then the best approach is to replace the first implication operator with a fusion operator as

discussed previously, $a \ \#\#0 \ b \ \mid\rightarrow \ c$. This way, if either a or b fails, then the expression has a vacuous success. c is only tested if both a and b pass.

As a side note, if the non-overlapping implication operator is used, then the $\#\#1$ should be used instead of the $\#\#0$. So $a \ \mid\Rightarrow \ b \ \mid\Rightarrow \ c$ would be replaced by $a \ \#\#1 \ b \ \mid\Rightarrow \ c$.

3.6 Use an Upper Range Bound

A variation to the original property shown in code Examples 10 and 11 is to replace the unbounded range with a fixed upper limit. Once the upper bound is reached, the $a2/c2$ (Figure 9) property expression will return with a pass. The $a2/c2$ property express will still fail on the first $a2$ pass/ $c2$ fail, just like the unbounded model shown in Figure 10.

```

property prop_16a;
  int v_cnt;
  @(posedge clk) $rose(a) |-> b[*0:8] \#\#1 c |-> d;
endproperty:prop_16a

property prop_16b;
  int v_cnt;
  @(posedge clk) $rose(a) |-> first_match(b[*0:8] \#\#1 c) |-> d;
endproperty:prop_16b

example_16a: assert property (prop_16a);
example_16b: assert property (prop_16b);

```

Example 16 – Chained Implication with Bounded Repetition Range in Antecedent

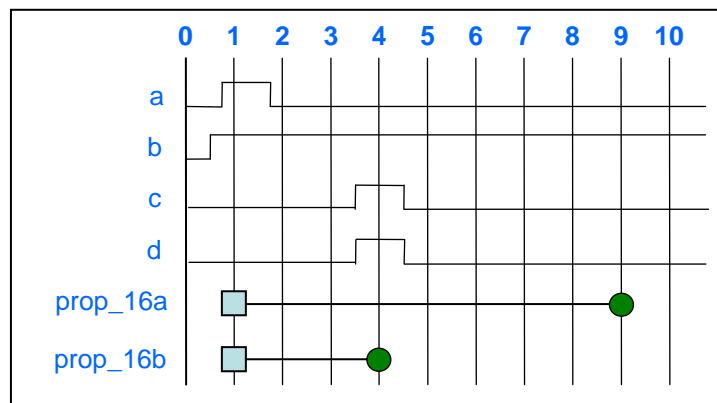


Figure 17 – Bounded Range With and Without a `first_match`

Providing an upper bound can close the property expression giving it an ending time and therefore a point to respond with a pass. Perhaps more interesting would be to use a variable for the upper bound such as `b[*0:upper]`. Unfortunately, range values must be constants or constant expressions. However, with some hand-waving and some fancy coding, a variable

upper limit can still be applied. This variable could be passed into the property as an argument or the variable could just be accessed globally. The following code example, prop_17c shows one way to model a range with a variable upper limit

```

module foo;
  bit a, d, clk;
  int upper;

  property prop_17a;
    int v_cnt;
    @(posedge clk) $rose(a) |-> b[*0:$] ##1 c |-> d;
  endproperty:prop_17a

  property prop_17b;
    int v_cnt;
    @(posedge clk) $rose(a) |-> first_match(b[*0:$] ##1 c) |-> d;
  endproperty:prop_17b

  property prop_17c;
    int v_cnt;
    @(posedge clk) ($rose(a) && (upper != 0), v_cnt = 0) |->
      ((v_cnt < upper), v_cnt++)[*0:$] ##1 c |-> d;
  endproperty:prop_17c

  ap17a: assert property (prop_17a);
  ap17b: assert property (prop_17b);
  ap17c: assert property (prop_17c);

  initial begin
    upper = 8;
    . . .
  end

```

Example 17 – Variable Used to Set the Upper Range Bound

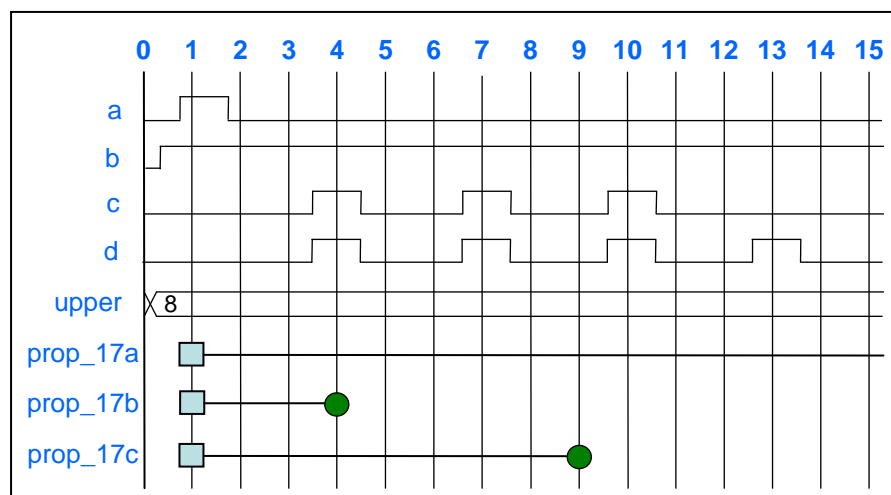


Figure 18 – Variable Upper Range Bound

Figure 18 shows the contrast between the original property expression (prop_17a), the property expression with the **first_match** for the second antecedent (prop_17b), and finally, the property expression with a variable max range (prop_17c). The bounded range property is modeled in this example so that it will exit once the range has been reached.

One additional variation on using a variable upper bound is to only test when the upper bound has been reached. The following code example provides this capability.

```
module foo;
  bit a, c, d, clk;
  int upper;

  property prop_18;
    int v_cnt;
    @(posedge clk) ($rose(a) && (upper != 0), v_cnt = 0) |->
      ((v_cnt < upper), v_cnt++)[*0:$] ##1
      (v_cnt == upper) ##0 c |-> d;
  endproperty:prop_18

  ap18: assert property (prop_18);

  initial begin
    upper = 8;
    . . .
  end
endmodule
```

Example 18 – Variable Used to Set Upper Range Bound, Only Test When Upper Bound is Reached

This property will test once for c and d only when the upper count has been reached. Note that the **first_match** operator is not used or needed for prop_17c or prop_18 because once v_cnt value is greater than upper, the repetition ends.

3.7 Allow Only the First Antecedent to be Vacuous

When using chained implications, there may be times when a designer will want the entire consequent of the first antecedent to be considered false if any of the chained antecedents are vacuous. Consider the chained property expression $a \mid\rightarrow b \mid\rightarrow c$. If a is false, then the expression is vacuously true. If a is true and b is false, the expression is still vacuously true. The following code example provides a method in which the final property expression will be vacuously true solely due to a.

```
property prop_19a;
  @(posedge clk) a |-> b |-> c ;
endproperty:prop_19a
```

```

property prop_19b;
    @(posedge clk) a |-> b;
endproperty:prop_19b

property prop_19c;
    @(posedge clk) prop_19a and prop_19b;
endproperty:prop_19c

example_19c: assert property (prop_19c);

```

Example 19 – Chained Implication Where Only First Antecedent Can be Vacuously True

In Example 19, `prop_19a` will return a vacuous success for either `a` or `b` as described in the preceding paragraph. But, the result of `property prop_19c` will create a property expression where only condition `a` could cause a vacuous success.

4 Conclusions

Using chained implications within a property expression for a concurrent assertion can be complex and confusing. This complexity can be compounded if the second antecedent contains an unbounded range. Trying to replace the single property containing chained implications with multiple properties and assertions does not provide an equivalent model of the original chained property expression.

This paper has shown the problems associated with unbounded ranges in a first-stage consequent second-stage antecedent of a property containing chained implication operators. The biggest issue is that an unbounded range in an antecedent will cause the property expression to never end, thus never returning a pass condition. This is confusing if there are chained implications and the unbounded range is in the consequent of the first implication and in the antecedent of the second implication of the chaining. The consequent of implications have an implied first-match, but that does not work as expected when chaining implications where unbounded ranges are used. In actuality, the first-match does work. The problem is that an unbounded range in an antecedent prevents the property expression from ever ending. If the property never ends, there is never a first-match pass.

There are three ways in which a property with an unbounded range in the antecedent can end. First, if the consequent ever fails, the property expression ends. Second, by applying the `first_match` operator to the antecedent containing the unbounded range, the implication operator will be limited to only matching the one antecedent pass to one consequent match. And third, applying an upper bound to the range in the antecedent provides an end point for the property expression to return a pass.

The right solution is dependent upon the design and the intent of the property. Knowing how unbounded ranges work in an antecedent will help in choosing the right solution.

4 References

- [1] “*The Art of Verification with SystemVerilog Assertions*”, book by Faisal I. Haque, Johathan Michelson, Khizar A. Khan. Published by Verification Central, copyright 2006, ISBN-13: 978-0-9711994-1-5
- [2] “*SystemVerilog Assertions Handbook*”, book by Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari. Published by VhdlCohen Publishing, copyright 2005, ISBN 0-9705394-7-9
- [3] “*IEEE 1800-2005 standard for the SystemVerilog: Unified Hardware Design, Specification and Verification Language*”, IEEE, Piscataway, New Jersey, copyright 2005. ISBN 0-7381-4811-3.
- [4] “*Being Assertive With Your X*”, paper by Don Mills. Published in the proceedings of SNUG San Jose 2004

5 About the Author

Mr. Don Mills has been involved in ASIC design since 1986. During that time, he has worked on more than 30 ASIC projects. Don started using top-down design methodology in 1991 (Synopsys DC 1.2). Don has developed and implemented top-down ASIC design flow at several companies. His specialty is integrating tools and automating the flow. Don works for Microchip Technology Inc. as an internal SystemVerilog and Verilog consultant. Don is a member of the IEEE Verilog and System Verilog committees that are working on language issues and enhancements. Don has authored and co-authored numerous papers, such as “SystemVerilog Assertions are for Design Engineers Too!”, “RTL Coding Styles that Yield Simulation and Synthesis Mismatches”, and “Standard Gotchas” papers on the gotchas of Verilog and SystemVerilog. Copies of these papers can be found at www.lcdm-eng.com. Mr. Mills can be reached at mills@lcdm-eng.com or don.mills@microchip.com