```
property p_mem_write_read (mem_read, mem_write); // untyped arguments
        int v_address, v_data;        } // local variables
        logic v_parity;               } // in assertion variable declaration
        @ (posedge clk)
          $rose(go) |=> request ##[1:5] rdy
                        ##0 q_memory_write(v_address, v_data, v_parity, write)
                        ##[1:$] mem_read && address==v_address
                       ##1 data==v_data && parity==v_parity;
 endproperty : p_mem_write_read
 ap_mem_write_read : assert property (p_mem_write_read(read, write));
endmodule : Ch2_formal
```

Figure 2.7.5-2 Property using local variables (Ch2/Ch2_formal.sv)

The property *p_mem_write_read* states that upon a new *go*, a *request* is followed after a delay of one to five cycles by *rdy.* Following the *rdy* is the sequence *q_memory_write* that stores, upon a memory write the values of the memory access into variables local to that sequence. These include the *address*, *data*, and computed *parity* that is written into the memory. That information is later used upon the occurrence of a *read* to compare the memory data against what was previously written. Since the owner of these variables (*v_address, v_data, v_parity, write*) is the property *p_mem_write_read*, the consumer sequence *q_memory_write* must return the collected information back to the property variables. Thus, the property local variables get connected to the *q_memory_write* sequence variable *lv_address, lv_data, lv_parity* when the sequence is called. The sequence must then provide that information to the calling host, the property in this case. To accomplish this intimate link between the producer of the variables and the consumer of those variables, the consumer must declare those variables as `local` with a type and a direction.

Following the write, the property waits for a memory read at the previously stored address that was collected via the sequence *q_memory_write* that collected the write information into the variables local to the property.

2.7.6 No empty match in local variables assignments (rule 5)

📖 **Rule:** The subsequence to which a local variable assignment is attached must not admit an empty match. For example, consider the sequence following sequence:

```
a ##1 (b[*0:1], v_addr=0)          // 💣 Illegal because sequence equivalent to:
a ##1 (b[*0], v_addr=0)   or   a ##1 (b[*1], v_addr=0)
        ←---- Empty Match  →  // v_addr cannot be assigned in an empty match
```

However, the following is acceptable because there is no empty match.

```
( (a ##1 b[*0:1]), v_addr=0)  // 👍 ✓OK because sequence is equivalent to:
( (a ##1 b[*0]), v_addr=0)    or ( (a ##1 b[*1)], v_addr=0))
   (  (a), v_addr=0))    or ( (a ##1 b[*1)], v_addr=0))
  ←  Not empty  →              ←  Not empty        →
```

2.7.7 Local variable must be written once before being read (rule 6)

📖 **Rule:** A local variable must have a value assigned to it through initialization or assignment prior to being read. Consider the following:

```
sequence q_test (     // ch2/2.7/auto_var.sv
     local inout int lv_count);
     (a, lv_count+= 1); //
 endsequence : q_test

 property p_test;
     int v_c;  //.
     q_test(v_c);
 endproperty : p_test
```

Uninitialized local variable v_c is getting passed to an input type formal 💣

```
property p_test_OK;
    int v_c;
    (1, v_c=0) ##0 q_test(v_c) // ✓✓ v_c is uninitialized, value read in q_test
endproperty : p_test_OK
```

2.7.8 Variable is unassigned if not flowed out (rule 7, 10)

🕮 **Rule:** Local variables formal arguments of direction **input** and local variables declared in an *assertion_variable_declaration* do not flow out of the sequence in which they are used. Thus, such local variables become unassigned and do not flow out to the calling sequence. In addition, a local variable in an *assertion_variable_declaration* is not visible by other sequences or properties. For example:

```
sequence q_lv_no_flow_out(local input int lv_count);
  int v_data;
  `true ##1 (b, lv_count+=10, v_data=0) ##1 j==0; // 💣 Illegal,q_top. j is not visible
endsequence :  q_lv_no_flow_out


sequence q_top;
  int v_ct, j=0; // j is local
  (a,  v_ct=0)
  ##1 q_lv_no_flow_out(v_ct) //passing v_ct to sequence.
                               // Variable not flowed out. of q_lv_no_flow_out
  ##1 v_ct ==10  // v_ct is actually == 0 since formal argument not flowed out of the sequence
 ##1 v_data==0; // 💣 Illegal, v_data of sequence q_lv_no_flow_out cannot be referenced
endsequence :  q_lv_no_flow_out
```

@ entry, *lv_count=v_ct* (which is 0)
@ end point, *lv_count ==10, v_data ==0.*
@ exit of sequence, *lv_count and v_data*
do not flow out to q_top; they re local to the sequence

2.7.9 Local variables in concurrent and, or, and intersect threads (rule 14)

🕮 **Rule:** When local variables are used in sequences that have concurrent threads, it is possible that the values of the local variables may not be assigned, or assigned by one thread and not the other, or assigned in both threads at the same or at different cycles with different values. Concurrent threads in sequences occur with the use of the **and**, **or**, and **intersect** operators. *[1] In general, there is no guarantee that evaluation of the two threads results in consistent values for the local variable, or even that there is a consistent view of whether the local variable has been assigned a value. Therefore, the values assigned to the local variable before and during the evaluation of the composite sequence are not always allowed to be visible after the evaluation of the composite sequence.* Below are some examples of local variable flow.

2.7.9.1 Variables assigned on parallel "or" threads

🕮 **Rule:** When two sequences are ORed, each sequence produces its own thread that can get carried through to the next step, such as an end point (no more continuity), concatenation with another sequence, or as antecedent to a consequent. For example,

```
(sequence1 or sequence2) ##1 seq3                    // is equivalent to
(sequence1 ##1 sequence3) or (sequence1 ## sequence3)      // two threads
```

Note that multithreaded sequence do occur with range operators. For example:

```
 (a ##[1:2] b)  // is equivalent to
 (a ##1 b) or (a ##2 b).
  Also, (a ##1b[*1:2]) // is equivalent to:
   (a ##1 b[*1]) or (a ##1 b[*2]).
```

If the sequence makes assignments to local variables, then each of the sequence involved in the ORing carries its own individual and separate copy of the local variables. Those separate copies of the local variables are carried all the way through each of the ORed sequence thread. This concept is shown graphically in Figure 2.7.9.1.

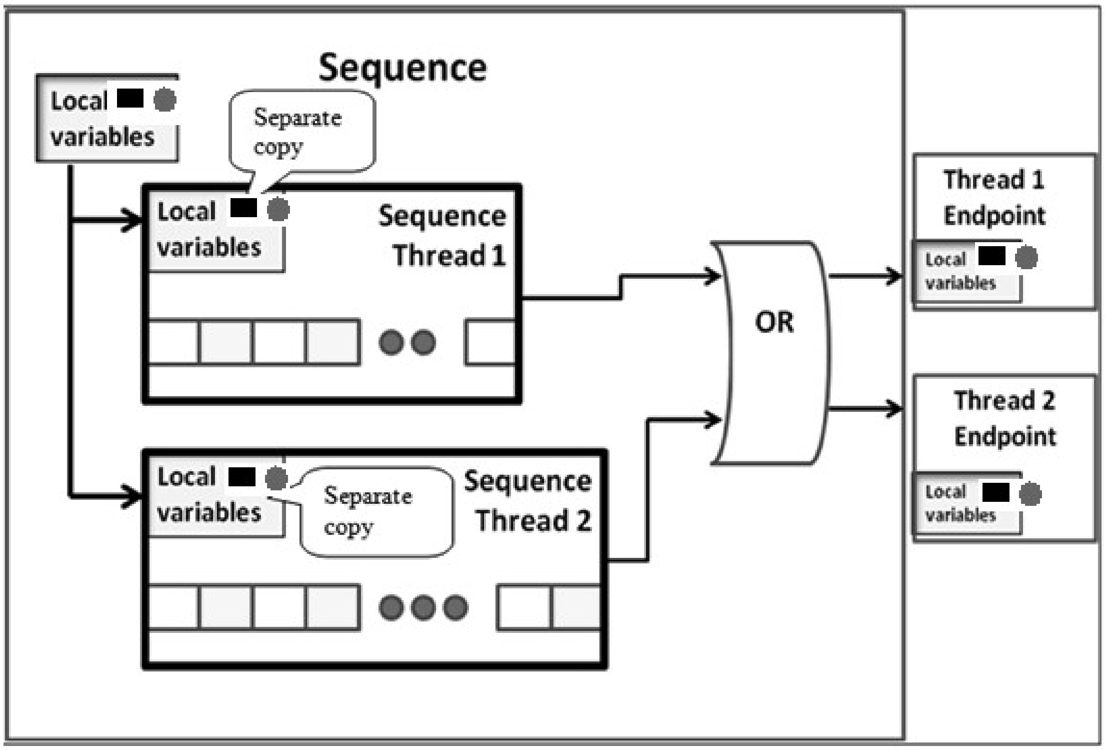

Figure 2.6.8.1 Oring of Two Sequences Maintain Individual Copies of the Local Variable Throughout the Lifetime of Each Thread. Threads exits separately

Each matching thread of an `or` operand continues as a separate thread; that matching thread carries with it its own copy of the local variables, which flow out of the composite sequence and out of the implication operator (i.e., |->, |=>). Thus, in summary, the local variables in properties with an antecedent and consequent are handled in the following manner:
1) If a multithreaded antecedent updates a local variable within its thread, each thread carries its own copy of the local variable.
2) If that property local variable is used in the consequent, the copy of that local variable is carried for each of those threads.
Consider the following property:

```
property p_abv;
        bit v;
        (a, v=1) or (b, v=0) |=> v==1;
endproperty : p_abv
```

The antecedent of this property has two threads (a, v=1) and `(b, v=0)`; each of those thread carries its own individual copy of the local variable. In the consequent, each antecedent thread must be evaluated with consequent. In this case, the individual copy of the local variable for each thread is carried into the consequent when that thread is evaluated. Thus for thread (a, v=1) the v in the consequent is the copy used in that thread. Similarly, for thread (b, v=0) the v in the consequent is the copy used in that thread.
The equivalent property to `p_abv` is the following:

```
property p_abv_eqv;
        bit v1, v2;
        ( (a, v1=1'b1) |=> v1==1)  and
        ( (b, v2=1'b0) |=> v2==1);
endproperty : p_abv_eqv
```

In the above example, if `a==1'b1`, and `b==1'b1`, then the property will fail.

🕮 **Rule:** There are strict rules on the flowing out of the local variable copied into each of the ORed sequences. *[1] A local variable flows out of the composite sequence if it flows out of each of the operand sequences,* as is the case for the above example. *If the local variable is not assigned before the start of the composite sequence and* ***it is assigned in only one of the operand sequences, then it does not flow out of the composite sequence***. For example:

```
sequence q_no_flow_out;  // ch2/2.7/or_multi.sv
    int v_x, v_y;
    ((a ##1 (b, v_x = data, v_y = data1) ##1 c) or // v_x and v_ y assigned
    (d ##1 (`true, v_x = data) ##0 (e==v_x))) // v_x assigned,  v_ y unassigned
              // Thus, v_x flows out, and  v_y does not flow out
    ##1 (v_y==data2); // 💣 Local variable v_y referenced in expression where it does not flow.
/ Illegal: v_y cannot be read in thread 2 because it was not uninitialized
/ in the thread or during the initialization of the local variable at declaration.
endsequence : q_no_flow_out

sequence q_flow_out1; // OK
    int v_x, v_y;
    ((a ##1 (b, v_x = data, v_y = data1) ##1 c) or  // v_x and v_ y assigned
    (d ##1 (`true, v_x = data, v_y=0) ##0 (e==v_x)))  // v_x and v_ y assigned
    ##1 (v_y==data2);
endsequence : q_flow_out1

sequence q_flow_out; // 👍
  int v_x, v_y;
  ((a ##1 (b, v_x = data, v_y = data1) ##1 c)or   // Thread 1:  v_x and v_ y assigned and flow
  out
  (d ##1 (`true, v_x = data) ##0 (e==v_x)))          //  Thread 2:  v_x assigned
                                                 // Thus, v_x flows out, and  v_y does not flow out
  ##1 (v_x==data2);      // v_ y unassigned, but is not used in the subsequent sequence ##1
  (v_x==data2)
          // That subsequence only needs v_x,  which does flow out since it was initialized.
  endsequence : q_flow_out
```

2.7.9.1.1 first_match(seq1 or seq2)

🕮 **Rule:** As mentioned above, the ORing of two sequences produces two threads, each with their own individual copies of the local variables. A **`first_match`** of an OR of two sequences (i.e., ***`first_match(`****`seq1`* ***`or`*** *`seq2)`*) causes the match of its operand sequence to be minimal in length, but it does not forbid multiple matches of the same minimal length. As long as all the matches are of the same minimal length, they can have different valuations of the local variables. If an evaluation of **`first_match`** of a sequence has a double match of the same length, these two matches could have different values for the local variables at the end points. For example:

```
int k=1;
property p_abvFM;  // ch2/2.7/match_abc.sv
        bit v;
      first_match((a, v=1) or (b, v=0)) |=> v==k;
endproperty
```

`Property p_abvFM` results in 2 matches if `a==b==1` since each thread is a match. That example results in a failure of the property when `a==b==1` because one thread will succeed, and the other will fail.

Guideline: In sequences used as antecedents, ensure a unique first_match.

Design Example:
Given signal `a` and signal `b,` `b` must be high (consecutively or non-consecutively) at least `n` number of times between `rose(a)` and `fall(c)` `n` is defined in a module variable called "count".

The following solution guarantees a unique first_match:

```
property p_ab_count_OK;   // ch2/2.7/pulsesOK.sv
              int v;
              ($rose(a), v=0) |->
                  first_match (
                                 (  b || $fell(c)[->1] ##0
                                   (( b, v=v+1'b1) or !b && $fell(c))
                                 )[*1:$] ##0 $fell(c)
                               )
                  |-> v >=count;
endproperty
```

Note: The following sequence may result in <u>non-unique first_matches, and that is problematic</u>.

```
    first_match ( ##[1:2]
                      (  b || $fell(c)[->1] ##0
                             ((b, v=v+1'b1) or !b && $fell(c))  )[*1:$] ##0 $fell(c) )
```

This is because there is one thread starting at ##1, and the other at ##2. If at ##1 and at ##2 b==1 and c==1 (i.e., no fell(c), and if the match occurs at a later cycle then each of those two threads will have a match with a fell(c). However, an error will occur because each of those two threads that matches will have different values for their local variable, assuming that some b's occurred before the fell(c). .

2.7.9.2 Variables assigned on parallel “and” “intersect” threads

🕮 **<u>Rule:</u>** *[1] In the case of* `and` *and* `intersect`*, a local variable that flows out of at least one operand shall flow out of the composite sequence unless it is blocked. A local variable is blocked from flowing out of the composite sequence if either of the following statements applies:*

1) The local variable is assigned in and flows out of each operand of the composite sequence, or

2) The local variable is blocked from flowing out of at least one of the operand sequences.

The value of a local variable that flows out of the composite sequence is the latest assigned value. The threads for the two operands are merged into one at completion of evaluation of the composite sequence.

When two sequences are ANDed or INTERSECTed, each sequence produces its own thread. However, unlike the ORing of two sequences, <u>the **`and/intersect`** of two sequences produces a single end point at the termination of the threads</u>. This means that the two threads merge into a single starting point; this contrasts to the Oring of two sequences where each thread is carried separately to the next sequence.

If the sequence makes assignments to local variables, then each of the sequence involved in the ANDing or INTERSECTing carries its own individual and separate copy of the local variables. However, only <u>those local variables that are NOT assigned in both threads flow out of the sequence</u>. This concept is shown graphically in Figure 2.7.9.2. Thus, <u>it is illegal to have the same variables being assigned in each of the threads</u> <u>and have that variable flow out of the sequence</u> because the outputs of these operators produce a single end point with updated values for the variables.

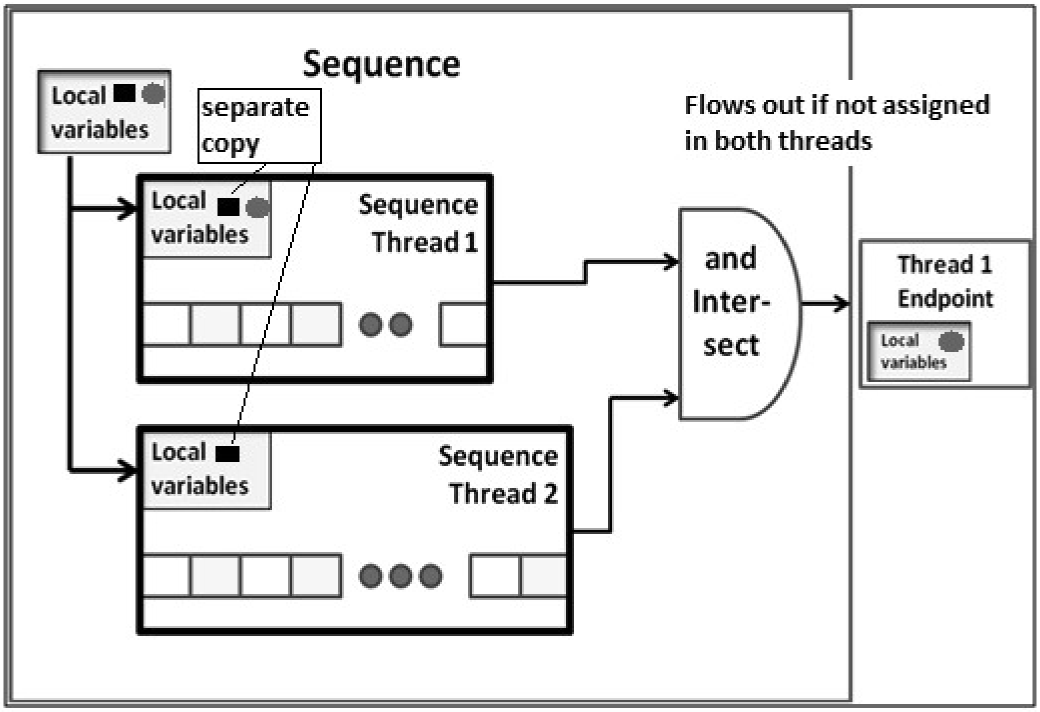

Figure 2.7.9.2 ANDing / INTERSECTing of two sequences maintain individual copies of the local variable throughout the lifetime of each thread. Threads combine into a single exit point

The following demonstrates various cases of this rule.

v is assigned here

v is NOT assigned here

v flows out

```
  sequence my_and;
   logic v;
//      ← my_and_part1 →     ← my_and_part2 →
     ( ((a, v=x) ##1 b) and (c ##1 d)        )

    ##1 e==v; // ✓ OK since v assigned ONLY in
              // one thread of the 2 threads of the and operator
  endsequence : my_and

     sequence q_no_out_flow(v);
        (
         ((a, v=x) ##1 b) intersect
        (c ##1 (d, v=y))
        )
        ##1 v==1;  // 💣 v is blocked from flowing out
             // because it is common to both threads
    endsequence : q_no_out_flow

  property p_and_legal;
     int x,y;
     ((a ##1 (b, x = data, y = data1) ##1 c)  and
      (d ##1 (`true, x = data) ##0 (e==x))
     ) ##1 (y==data2)
      // |=> x==1; // 💣Local variable x referenced in expression where it does not flow.
        |=> y==1; // ✓  local variable y is not assigned in both threads.
  endproperty : p_and_legal
```