1.0 Local Variables Flow-in for and/or/intersect/implies

http://www.eda-stds.org/svdb/view.php?id=3195

1.1 PROBLEM

Ref: IEEE 1800-2009, 16.10 Local Variables

Section 16.10 of the LRM provides several restrictions in the flow out of local variables when the **`and,or, intersect`** operators for sequences / properties are used. However, there is a need to have local variables being flowed-in concurrent sequences/properties defined within a property statement . Specifically, one should be able to set and read local variables in one sequence or property (e.g., LHS), and read those local variables in another sequence / property (e.g., RHS). For example (*simple model for demonstration of the issues, but a more complex is provided in section 1.2about the motivations*). This model shows the clear separation between cause and effect.

```
property P_illegal; // illegal in IEEE 1800-2009
     int v_t1;
     ((a, v_t1=data) ##1 b) implies   // cause, v_t1 can be read in LHS
     (##1 d==v_t1);                   //effect, v_t1 cannot be read in RHS
endproperty : P_illegal
ap_Ptop: assert property(@ (posedge clk) P_illegal);
```

The property **is illegal** because in the consequent `(##1 d==v_t1,)` the local variable `v_t1` is referenced in expression where it does not flow-in.

1.1.1 Repeat copy into multiple local variables solution

One solution that does not express the concept of cause / effect (for this example):

```
property P_legal_BUT;
     int v_t1, v_t2;
     ((a, v_t1=data) ##1 b) implies   // antecedent is the "cause"
     ((1, v_t2=data) ##1 d==v_t2);    // consequent is the effect.
            // Must repeat the writing of local variables in consequent,
            // thus mixing the "cause" / "effect", and that separation needed for formal verification.
endproperty : P_legal_BUT
ap_Ptop: assert property(@ (posedge clk) P_legal);
```

1.1.2 Temporary save value into module variable from sequence match item

1. At desired cycle "n" in the antecedent, **save object into module variable**.
2. At next cycle (i.e., "n+1), in the consequent property, copy the value of that module variable into the **local variable** of the concurrent sequence/property . That is very essential since another start of the property could write a new value into the module variable.
3. Use a task (with formal arguments -- preferred) to do the writing into the module variable. That local variable can then be read during that consequent sequence.

```
int vdt;
  task wr(int v);
    vdt = v;
  endtask : wr
property P_ok;
    int v_t1;
    ((a, wr(data)) // @t, copy data to variable vdt
      ##1 b) implies
    (##1 (1, v_t1=vdt) // @t+1 Save the copied variable in previous cycle to local variable
      ##2 d==v_t1);
  endproperty : P_ok
  ap_Ptop2: assert property(@ (posedge clk) P_ok);
```

1.2 MOTIVATIONS FOR LOCAL VARIABLES FLOW-IN

It was mentioned that for that simple model the implication operator can be used as an alternative because the local variables flow out of antecedents. However, there are several motivations to allow the flow-in of local variables into concurrent property statements / sequences that are ANDed or ORed. The proposal would allow for this with some restrictions that can be resolved at compilation time rather than during elaboration or run-time (*see section XXX).*

1.2.1 Formal verification for completeness tests

This capability allows for the description of operational properties where data values (e.g,, value of an addr_bus) must be saved in one property statement (typically in an antecedent) and then compared in a concurrent property statement (typically in a consequent). Allowing this coding style is needed for completeness check. See "*Complete Functional Verification*" Joerg Bormann http://fmv.jku.at/fmcad09/slides/bormann.pdf

1.2.2 More easily write assertions where concurrent processes are used

From a linguistic viewpoint, SVA currently supports the definition of concurrent sequence s/properties. However the language does not support the common access of local variables from within those concurrent sequence /properties. The use of concurrent sequences /properties facilitates the writing of assertions from a definition initially specified in a timing diagram.

Consider this example:
With a "go" the machine performs a one word transfer into memory from a bus interface. Machine has pipeline registers. We have a timing diagram of the transactions, as shown in Figure 1.2.2. In this transaction, there are four concurrent operations:
1) The *go* operation that may have information about the address of where to store the data
// call this P_go
2) The CPU DMA control stuff *// Call this property P_cpu*
3) The bus interface that feeds the data *// Call this property P_if*
4) The data transfer into the memory (maybe some cache involved) *// Call this property P_mem*
In straight SystemVerilog, one can use concurrent tasks with the fork/join; each task has the capacity to write/read module variables. For SVA, I can make use of concurrent sequences/properties to write this assertion as:

```
property P_machine;
  automatic int v_data;  // holds data from bus to be sent to memory.
  automatic int v_addr;  // holds the size of the dma transfer
    P_go implies P_cpu and P_if and P_mem;
endproperty :  P_cpu_dm_bus
```

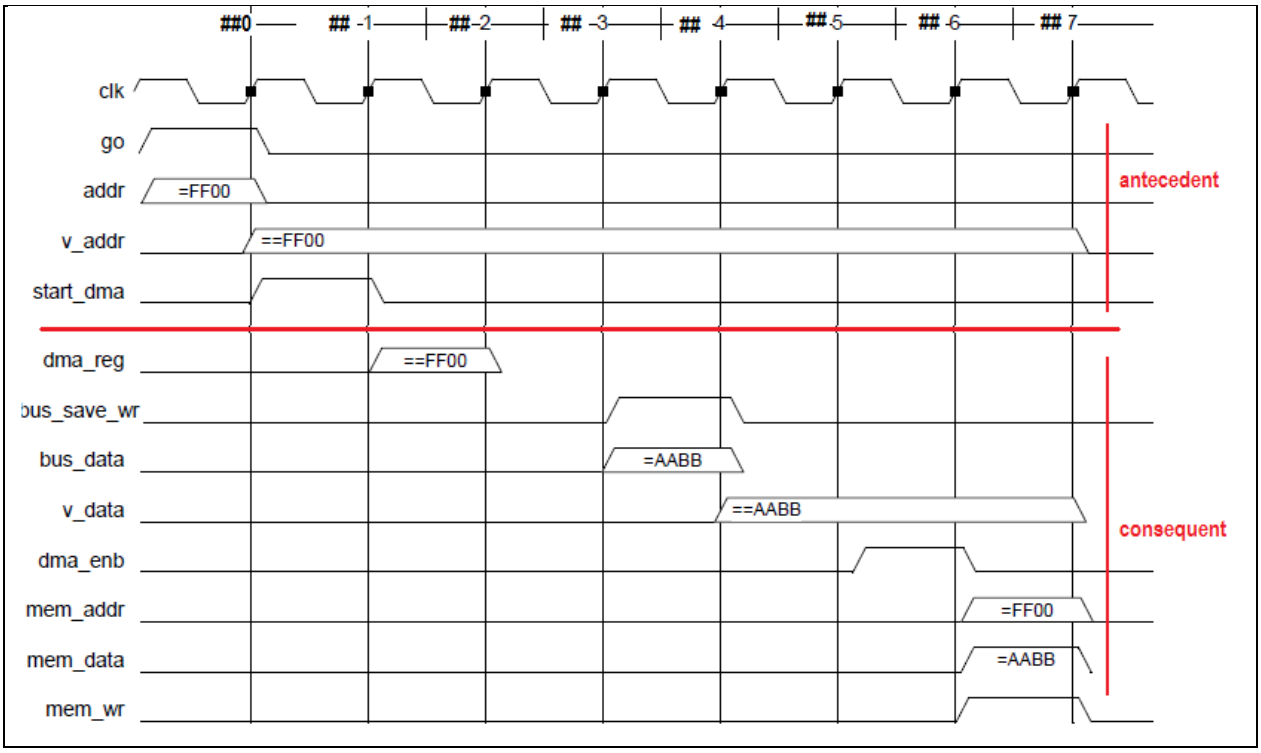

Figure 1.2.2 Timing Diagram for Machine Example

Expanding this hypothetical machine and knowing that this machine has pipeline registers, I can write the property using a style that expresses processes rather than sequential statements.

```
module  m;
// values initialized for quick verification of model
 int addr=16'hFF00,  bus_data=16'hAABB, mem_data=16'hAABB, mem_addr=16'hFF00, dma_reg=16'hFF00;
 logic clk=1, go=0, start_dma=1, dma_enb=1, mem_wr=1, bus_save_wr=1;
property P_machine;
  automatic int v_data;  // holds data from bus to be sent to memory.
  automatic int v_addr;  // holds the address of the memory
    ((go, v_addr=addr) ##1 start_dma)   implies   // P_go,  addr is held for 1 cycle in module
    (##2 dma_reg==v_addr  )    and          // P_cpu, the dma saves this address
    (##[3:4] (bus_save_wr,  v_data=bus_data)) and // p_if, interface data save
   (##[5:6]  dma_enb ##1
              mem_addr==v_addr && mem_data==v_data  && mem_wr); //  P_mem
endproperty :  P_machine
```

There is no question that this particular model can be expressed in a style that emphasizes sequential operations rather than concurrent processes. Specifically:

```
module  m; // This model compiles and simulates OK in IEEE 1800-2009
// values initialized for quick verification of model
 int addr=16'hFF00,  bus_data=16'hAABB, mem_data=16'hAABB, mem_addr=16'hFF00, dma_reg=16'hFF00;
 logic clk=1, go=0, start_dma=1, dma_enb=1, mem_wr=1, bus_save_wr=1;
property P_machine;
  int v_data;  // holds data from bus to be sent to memory.
  int v_addr;  // holds the address of the memory
    (($rose(go), v_addr=addr) ##1 start_dma)   |->  // P_go, addr is held for 1 cycle in module
    ##2 dma_reg==v_addr              // P_cpu, the dma saves this address
    ##[1:2] (bus_save_wr,  v_data=bus_data)  // p_if, interface data save
   ##[1:2]  dma_enb ##1
               mem_addr==v_addr && mem_data==v_data  && mem_wr; //  P_mem
endproperty :  P_machine
  aP_machine: assert property(@ (posedge clk) P_machine);
  initial forever #5 clk=!clk;
  initial begin
           @ (posedge clk) go<=1'b1;
           end
  endmodule : m
```

However, the point of this proposal is that **the language must be flexible enough to allow its use in a variety of ways, particularly if it adapts to a style that eases the verification of completeness in formal verification .**

There might be other cases where a user want to use local variable used in sequences/properties that are specified with "**and/or/intersect/implies**" operators. Section 16.10 goes to great lengths to explain the flow out of local variable in such cases. But I strongly believe that SVA needs to have local variables must **flow in** (or flow into) property/sequence statements, with the restrictions identified in section 1.3.

Currently, the only workaround in emulating this capability is to copy the desired value into a module variable using a task from within the sequence match item using a task or function, and then ping-ponging the value of that module variable into the local variable of the other property. See property *P_ok* in my 1.1.2. Again, formal verification needs this feature to ease the computation of completeness.

1.3 PROPOSED SOLUTION

In section 16.10 add the following:

1. Allow, as an option, the use of the reserved word **automatic** in the definition of local variables within properties. *I did not want to create another reserved word!*
2. The use of this keyword identifies that this local variable can be set once, and only once, but it can be read anywhere from within the property statement.
3. An assertion statement that uses automatic local variables must be singly clocked.
   The testing of the singly clocked assertion statement can be done at compile time. There are complexities when multiclocked assertion statements are used, as the checking of this rule would have to be done at run time. However, restricting the use of the automatic local variable to singly clocked assertion statement reduces the complexity. IEEE 1800-2009 currently requires the checking of writing/reading of local variables for compliance to the flow out rules in 16.10.
4. The local variable shall not be read in the same clock cycle it is written.
   [Ben] I believe that there would be a race condition because one property can write a value to the local variable while a concurrent property reads the property. For example:
   property P_race;
   automatic int v;
   (a, v=data) ##[1:3] b) implies (c==v ##2 d):
   endproperty :P_race
5. This **automatic** variable can be initialized in its declaration.
6. If this **automatic** variable is read before it is initialized, its value shall have its default value based on its type.

Example:

```
property P_proposed;
    automatic int v_t1;
    ((a, v_t1=data) ##1 b) implies  // cause
    (##1 d==v_t1);                  // effect, can now read v_t1
endproperty : P_ proposed
ap_Ptop: assert property(@ (posedge clk) P_ proposed);
```