SV3.1a Assumptions

Surreendra Dudani
Nov. 14, 2003

Synopsys, Inc.
Property Assumptions

\begin{verbatim}
assume property ( property_spec ) ;
\end{verbatim}

• Allow properties to be considered as assumptions for formal and dynamic simulation

• Tools must constrain the environment such that a property holds

• No obligation for tools to prove the property

• Assumptions may be used as embedded or as declarative
Request - acknowledge protocol

• When reset asserted, keep req de-asserted
• req can be raised at any other time
• req once raised, stays asserted until ack is asserted
• In the next clock cycle both req and ack must be de-asserted
Assumptions governing req

property pr1;
    @(posedge clk) !reset_n |-> !req;
    // when reset_n is asserted (0), keep req 0
endproperty

property pr2;
    @(posedge clk) ack |=> !req;
    // one cycle after ack, req must be deasserted
endproperty

property pr3;
    @(posedge clk) req |-> req[*1:$] ##0 ack;
    // hold req asserted until and including ack asserted
endproperty
Assertions governing ack

property pa1;
    @(posedge clk) !reset_n || !req |-> !ack;
endproperty

property pa2;
    @(posedge clk) ack |=> !ack;
endproperty
Assume req - Assert ack

\textbf{a1: assume property (pr1);} \\
\textbf{a2: assume property (pr2);} \\
\textbf{a3: assume property (pr3);} \\

\textbf{s1: assert property (pa1)} \\
\text{else } \texttt{display("
ack asserted while req is still deasserted");} \\
\textbf{s2: assert property (pa2)} \\
\text{else } \texttt{display("
ack is extended over more than one cycle");}
Assumptions with biasing

assertion_expression ::= expression |
expression dist { dist_list }

• Needed for driving random simulation

• Biasing provides a weighted choice to select the value for free variables

• Alignment with constraint block syntax

• For assertions and formal, dist converted to inside

• Properties must hold with or without biasing
Assume req - Assert ack with biasing

a0: assume property @(posedge clk) req dist {0:=40, 1:=60} ;
a1: assume property (pr1);
a2: assume property (pr2);
a3: assume property (pr3);

s0: assert property @(posedge clk) req inside {0, 1} ;
s1: assert property (pa1)
    else $display("\n ack asserted while req is still
deasserted");
s2: assert property (pa2)
    else $display("\n ack is extended over more than
    one cycle");
Sequential constraints in constraint block

constraint_block ::= property_instance ; | ..... 

• Extend constraint block to allow property instances
• Properties become assumptions, and automatically create boolean constraints
• Boolean constraints from properties solved with other boolean constraints, whenever randomize function called
• Procedural control over randomization as before
Example: constraints using properties

• When signal X is set to 1, then on the next cycle X must be reset to 0, and remains stable for four cycles, at the end of which signal Z must be set to 0 for at least one cycle.

• When signal Y is set to 1, then on the next cycle it is reset to 0, and remains stable for two cycles at which point signal Z must be set to 1 for at least one cycle.
Code using sequential constraints

property pA;
    @(posedge clk) X |=> (!X)[*4] ##0 !Z;
endproperty
property pB;
    @(posedge clk) Y |=> (!Y)[*2] ##0 Z;
endproperty

class drive_XYZ_class;
    rand bit X;
    rand bit Y;
    rand bit Z;
    constraint pAB_Z {pA; pB;}
endclass
Randomizing sequential constraints

program P(input bit clk, output bit x_port, y_port, z_port);
   drive_XYZ_class XYZ_driver = new();
   initial
      forever begin
         @(negedge clk) void’XYZ_driver.randomize();
         x_port = X;
         y_port = Y;
         z_port = Z;
      end
   ...
endprogram
class drive_XYZ_class;
    bit [2:0] A_state; bit [1:0] B_state;
    rand bit X,Y,Z;
    constraint AB_Z {
        (( |A_state[1:0]) == 1'b1 ) => (X == 1'b0);
        (A_state == 3'b100) => (Z == 1'b0) && (X == 1'b0);
        (B_state == 2'b01) => (Z == 1'b1);
        (B_state == 2'b10) => (Z == 1'b1) && (Y == 1'b0);
    }
    // functions on the next slide
endclass
Functions for combinational constraints

function void new(); // initialize state variables
    A_state = 3'b0; B_state = 2'b0;
endfunction

function void post_randomize(); // advances state machine
    case (A_state)
        3'b000 : if (X == 1'b1) A_state = 3'b001;
        3'b001, 3'b010, 3'b011 : A_state = A_state + 3'b001;
        3'b100 : A_state = 3'b000;
        default : $display("bad A_state");
    endcase

    case (B_state)
        2'b00 : if (Y == 1) B_state = 2'b01;
        2'b01 : B_state = 2'b10;
        2'b10 : B_state = 2'b00;
        default : $display("bad B_state");
    endcase

endfunction
endclass
Randomizing combinational constraints

program P(input bit clk, output bit x_port, y_port, z_port);
    drive_XYZ_class XYZ_driver = new();
    initial
        forever begin
            @(negedge clk) void’XYZ_driver.randomize();
            x_port = X;
            y_port = Y;
            z_port = Z;
        end
    ...
endprogram
Path to dead end state

• In either implementation, the constraint solver can legitimately set X to 1 in a cycle, then two cycles later, set Y to 1.

• After two cycles the solver is unable to solve for Z, i.e., the system of constraints is inconsistent at that point

• The problem is due to under-constraining (an incomplete set of constraints)

• Adding an additional constraint, dead end state can be eliminated
Eliminating dead end state

- In either implementation, adding an additional constraint, dead end state can be eliminated

```vhdl
property pC;
    @(posedge clk) X |-> ##2 !Y;
endproperty
```
Summary

- Simple syntactic extensions
- \textit{assume} construct provides assumptions for formal analysis
- \textit{assume} construct with biasing provides automatic random simulation
- Properties as sequential constraints simplify coding for randomizing complex sequences
- Difficulties and issues of dead end state exists for both combinational and sequential constraints