1.1 Using properties in the constraint block

The extensions proposed in Section 1.2 provide an easy mechanism for the application of assumptions. Along with the biasing, this mechanism is adequate for simple cases. For simulation, tools can determine free and state variables of the assumptions, apply randomization to the free variables, and drive inputs to the design such that all assumptions for the design are valid.

Often times, for more complicated cases, users need a better control over what gets randomized and when inputs to the design are driven. This must be carefully considered based on how outputs of the design settle in time relative to the clock. System Verilog provides a feature to specify boolean constraints using the **constraint** construct. The **constraint** construct is supported by random variable declarations and a function to randomize random variables using the boolean constraints. This functionality is encapsulated by the class definition which allows to package multiple constraint definitions, random variables representing the free variables in the constraints, and a randomize function that chooses values for the random variables. Refer to Section 12.4 of the System Verilog LRM for details.

This proposal extends the **constraint** construct by allowing instantiations of properties within the constraint block.

1.1.1 Syntax

constraint_declaration ::=

[**static**] **constraint** constraint_identifier **{** { constraint_block } **}** *// from Annex A.1.9*

constraint_block ::=

solve identifier_list **before** identifier_list **;**

| expression **dist {** dist_list **} ;**

| property_instance **;**

| constraint_expression

constraint_expression ::=

expression **;**

| expression **=>** constraint_set

| **if (** expression **)** constraint_set [**else** constraint_set]

constraint_set ::=

constraint_expression

| **{** { constraint_expression } **}**

constraint_prototype ::= [**static**] **constraint** constraint_identifier

extern_constraint_declaration ::=

[**static**] **constraint** class_identifier **::** constraint_identifier **{** { constraint_block } **}**

identifier_list ::= identifier { **,** identifier }

The syntax of constraint_declaration is extended by including property_instance in constraint_block.

1.1.2 Semantics

When a class containing constraint blocks is instantiated, the properties start like assertions. The expressions of the properties form boolean constraints. These boolean constraints are conjoined with all other boolean constraints specified in the class. Properties may use random variables declared in the class. When the randomize function is invoked, the constraints are solved by choosing the appropriate random variable values. At the next clock tick, properties advance forward according to the values obtained by solving the constraints and a new set of constraints is thus selected for the next call to the randomize method.

While properties act like assertions to ensure that they hold at every clock tick, they provide boolean con-

straints that must be solved at the time of randomization. Here, the users determine the appropriate time to randomize, using their knowledge of the design.

All other features available with constraint blocks such as constraint_mode ON/OFF apply to constraints inferred from properties.

1.1.3 Example of the constraint block

Using the same example in 1.2.5, the code below illustrates the use of constraint blocks.

```
program p(input bit ack, input bit clk, output reset_n, output bit req);

property pr1(req, ack);
   @(posedge clk) !reset_n |-> !req;
endproperty
property pr2(req, ack);
   @(posedge clk) ack |=> !req;
end property
property pr3(req, ack);
   @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty

class req_class;
   rand bit v_req = 0;
   constraint req_constr {
      v_req dist {0:=40, 1:=60};
      pr1(v_req, ack);
      pr2(v_req, ack);
      pr3(v_req, ack);
   }
endclass
initial begin
   reset_n = 0;
   @(posedge clk);
   reset_n = 1;
end

initial begin
   req_class drive_req = new();
   while (not_done) begin
      @(negedge clk);
      drive_req.randomize();
      // drive the output using drive_req
      req = v_req;
   end
......
......
end

endprogram
```

1.1.4 Example of equivalence between sequential constraints and combinational constraints

This example illustrates how a sequential constraint is equivalent to combinational constraint specification in System Verilog. For the same constraint problem, the two equivalent versions are presented.

In System Verilog, combinational constraints are specified in a class using the constraint block and random variables. To randomize the random variables, the class object is invoked with the randomize method. The ran-

domize method is expected to choose a value for each random variable by using the biasing specification, if provided by the user, and by ensuring that the combinational constraints are satisfied. However, the randomize method may return an error, if no value of at least one random variable can be chosen to satisfy the combinational constraints. This occurs if the user constraints are incomplete or incorrect, assuming that the design has no errors.

Typically, the values of random variables are used to drive the free inputs of the design. It may happen that the outputs of the design, as a result of applying the chosen values to the free inputs, make the combinational constraints inconsistent to solve at the next invocation of the randomize method. This situation is commonly referred as a “dead end state”.

In this example, it will be shown that either version, combinational or sequential, may enter a dead end state due to incomplete constraint specification.

1.1.5 Constraint problem specification

The constraints in this example are:

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

— if signal Y is set to 1, then on the next cycle it is reset to 0, and remains 0 for two cycles at which point signal Z must be set to 1 for at least one cycle.

1.1.6 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
```

Once constraints are written, the user needs to decide an appropriate point in the cycle to drive the inputs. Generally, the user would drive the inputs ahead of the arrival of the clock, such that the combinational logic driven by the inputs settle before the clock arrival. In this example, the user has chosen to randomize and drive the inputs at the mid-point of the cycle `(negedge clk)`.

```
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
```


One way to implement properties as constraints is to implicitly construct automata for the properties. Using the transition functions and states of the automata, combinational constraints are derived. The randomize method then internally uses the state values of the automata in solving for the random variables X,Y and Z with respect to the derived combinational constraints.

1.1.7 Code using combinational constraints

Without sequential constraints, the user could implement similar behavior using just combinational constraints and some state variables as follows:

```
class drive_XYZ_class;
   bit [2:0] A_state;
   bit [1:0] B_state;
   rand bit X;
   rand bit Y;
   rand bit 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);
   }
   // initialize state variables
   function void new();
      A_state = 3'b0;
      B_state = 2'b0;
   endfunction
   // In SV, post_randomize is called automatically immediately after a call to
   // randomize. post_randomize updates the state machine
   function void post_randomize();
      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

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
```

Somewhere in the interface to the design, the following properties must hold. These are important to check for the combinational coding as the sequences are manually coded and driven. These assertions ensure that the driving adheres to the intended sequences. In the previous case of the sequential constraints, the properties acting as constraints ensure that the sequences are property followed.

```
property pA;
   @(posedge clk) x_port |=> (!x_port)[*4] ##0 !z_port;
endproperty
property pB;
   @(posedge clk) y_port|=> (!y_port)[*2] ##0 z_port;
endproperty
// following assertions must hold at the clock
x_prop: assert property (pA);
y_prop: assert property (pB);
```

1.1.8 A path to dead end state

In either implementation, the constraint solver can legitimately set X to 1 in a certain cycle, then two cycles later, set Y to 1. After a further two cycles, however, the solver is unable to solve for Z, i.e., the system of constraints is inconsistent at that point. The simulation is said to be in a dead-end state.

The problem is due to under-constraining (an incomplete set of constraints). For example, by adding the following property as one of the constraints the dead end is eliminated:

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

