Generate construct in sequences and properties

Objectives:

Generate

Allow nested sequence and let declarations in sequence declarations, and property, sequence and let declarations inside property declarations.

Allow generate construct inside sequence and property declarations, but only over nested sequence and property declarations.

The use of keywords generate - endgenerate is mandatory.

A separate set of BNF non-terminal nodes is created, defining the construct allowed inside such generate blocks.

7.6 Declaring sequences

REPLACE

A **`sequence`** can be declared in
— A module
— An interface
— A program
— A **`clocking`** block
— A package
— A compilation-unit scope

WITH

A **`sequence`** can be declared in
— A module
— An interface
— A program
— A **`clocking`** block
— A package
— A compilation-unit scope
— Another **`sequence`**
— A **`property`**

REPLACE

Variables used in a sequence that are not formal arguments to the sequence are resolved according to the scoping rules from the scope in which the sequence is declared.

WITH

Variables used in a sequence that are not formal arguments to the sequence are resolved according to the scoping rules from the scope in which the sequence is declared. This also means that if a sequence or a property contains a declaration of a local variable, the variable is directly visible throughout its body and thus also in any sequence declared within. That variable can thus be read or assigned. The variable flow rules are verified once the declared sequences are instantiated.

INSERT before 17.6.1

The following example shows a local declaration of a sequence in another sequence:

```
sequence rule_1;
  sequence s_local;
     a ##1 b ##1 c;
  endsequence
@(posedge sysclk)
  trans ##1 start_trans ##1 s_local ##1 end_trans;
endsequence
```

This is equivalent to the following sequence:

```
sequence rule_1;
@(posedge sysclk)
  trans ##1 start_trans ##1 a ##1 b ##1 c ##1 end_trans;
endsequence
```

It is illegal to reference sequence declarations within another sequence or property from the outside of the enclosing sequence or property.

For example, the following reference to s_local is illegal:

```
sequence rule_2;
  trans #sequence #1 bad_start ##1 rule_1.s_local;
endsequence
```

In Table *Syntax 17-4*, and *A.2.10*

REPLACE

sequence declaration ::=
sequence sequence_identifier [**(** [tf_port_list] **)**] **;**
{assertion_variable_declaration} sequence_expr **;**
endsequence [**:** sequence identifier]

WITH

sequence declaration ::=
sequence sequence_identifier [**(** [tf_port_list] **)**] **;**
{sequence_or_generate_item}
sequence_expr **;**
endsequence [**:** sequence_identifier]

sequence_or_generate_item
    sequence_item
  | sequence_generate_region

sequence_item ::=
    assertion_variable_declaration
  | sequence_declaration
  | let_declaration

sequence_generate_region ::=
  generate { sequence_generate_item } **endgenerate**

sequence generate item ::=
    sequence_loop_generate_construct
  | sequence_conditional_generate_construct
  | sequence_item

sequence_loop_generate_construct ::=
  for (genvar_initialization **;** genvar_expression **;** genvar_iteration)
            sequence_generate_block

sequence_conditional_generate_construct ::=
  sequence_if_generate_construct
  | sequence_case_generate_construct

sequence_if_generate_construct ::=
  if (constant_expression **)** sequence_generate_block_or_null [**else**
sequence_generate_block_or_null]

sequence_case_generate_construct ::=
  case (constant_expression) sequence_case_generate_item { case_generate_item } **endcase**

sequence_case_generate_item ::=
  constant_expression { **,** constant_expression } **:** sequence_generate_block_or_null
  | **default** [**:**] sequence_generate_block_or_null

sequence_generate_block_or_null ::= sequence_generate_block | **;**

sequence_generate_block ::=
  sequence_generate_item
  | [generate_block_identifier **:**] **begin** [**:** generate_block_identifier]
      { sequence_generate_item }
    end [**:** generate_block_identifier]

INSERT

17.6.2 Generate constructs in sequence declaration

Generate constructs provide the ability for actual arguments to affect the sequence implementation, thus making sequences more flexible and generic. Since generate schemes are evaluated during the model elaboration all their control expressions should be known at the elaboration time.

Consider the following example.

```
sequence follows(a, b, n);
  generate
    if (n > 0) begin : seq
      sequence a_to_b;
        a ##1 !a[*(n-1)] ##1 b;
      endsequence
    end : seq
    else if (n == 0) begin : seq
      sequence a_to_b;
        a && b;
      endsequence
    end : seq
    else begin : seq
      sequence a_to_b;
        b ##1 !a[*(n-1)] ##1 a;
      endsequence
    end : seq
  endgenerate
  seq.a_to_b; //instantiate the generated sequence
endsequence
```

This sequence states that `b` should be asserted in `n` cycles after `a` has been asserted for the last time. If `n` is negative then `b` precedes the next occurrence of `a` by `n` cycles. E.g., `follows(read, write, 3)` will be expanded into `read ##1 !read[*2] ##1 write`, `follows(read, write, -3)` will be expanded into `write ##1 !read[*2] ##1 read`, and `follows(read, write, 0)` into `read && write`.

In the following example, the generate construct builds a concatenation (`##1`) of subsequences, each of which is of length 1 over a bit from a vector passed as argument to the top sequence definition. Compile-time message is used to indicate if the vector is just a scalar (see Compile-time user messages. Note to the editor please insert cross reference)

```
sequence s(vect);
   generate
      if ($bits(vect) == 1) begin : err $comp_error("not a vector"); end
      for (genvar i = 0; i < $bits(vect); i++) begin : Loop
          if (i==0) begin : Cond
             sequence t; vect[0]; endsequence
          end : Cond
          else begin : Cond
             sequence t; vect[i] ##1 Loop[i-1].Cond.t; endsequence
          end : Cond
      end : Loop
   endgenerate
   Loop[$bits(vect)-1].Cond.t;
endsequence
```

17.8 Manipulating data in a sequence

REPLACE

To access a local variable of a subsequence, a local variable must be declared and passed to the instantiated subsequence through an argument. The example below illustrates this usage.

```
sequence sub_seq2(lv);
  (a ##1 !a, lv = data_in) ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
sequence seq2;
  int v1;
```

```
  c ##1 sub_seq2(v1) ##1 (do1 == v1); // v1 is now bound to lv
endsequence
```

WITH

To access a local variable of a subsequence, a local variable must be declared and passed to the instantiated subsequence through an argument. The example below illustrates this usage.

```
sequence sub_seq2(lv);
  (a ##1 !a, lv = data_in) ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
sequence seq2;
  int v1;
  c ##1 sub_seq2(v1) ##1 (do1 == v1); // v1 is now bound to lv
endsequence
```

If `sub_seq2` is declared as a local sequence declaration within `seq2` then the local variable `v1` is visible inside `sub_seq2` and need not be passed as an argument, as shown in the following example:

```
sequence seq2;
  int v1;
  sequence sub_seq2;
    (a ##1 !a, v1 = data_in) ##1 !b[*0:$] ##1 b && (data_out == v1);
  endsequence
  c ##1 sub_seq2(v1) ##1 (do1 == v1);
endsequence
```

17.11 Declaring properties

REPLACE

A **`property`** can be declared in any of the following:
— A module
— An interface
— A program
— A **`clocking`** block
— A package
— A compilation-unit scope

WITH

A **`property`** can be declared in any of the following:
— A module
— An interface
— A program
— A **`clocking`** block
— A package
— A compilation-unit scope
— Another **`property`**

In 17.11.3

REPLACE

A named property can be instantiated by referencing its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. Like sequence declarations, variables used within a property that are not formal arguments to the property are resolved hierarchically from the scope in which the property is declared.

WITH

A named property can be instantiated by referencing its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. Like sequence declarations, variables used within a property that are not formal arguments to the property are resolved hierarchically from the scope in which the property is declared.
If a property is declared within another property, then the local variables declared in the enclosing property are visible in the locally declared property and thus need not be passed as arguments.

It is illegal to refer to locally declared properties of a property from the outside of the enclosing property.

In Table *Syntax 17-14*, and *A.2.10*

REPLACE

property_declaration ::=
property property_identifier [**(** [tf_port_list] **)**] **;**
{ assertion_variable_declaration }
property_spec **;**
endproperty [**:** property_identifier]

WITH

property_declaration ::=
property property_identifier [**(** [tf_port_list] **)**] **;**
{property_or_generate_item}
property_spec **;**
endproperty [**:** property_identifier]

property_or_generate_item ::=
property_item
| property_generate_region

property_item ::=
sequence_item
| property_declaration

property_generate_region ::=
generate { property_generate_item } **endgenerate**

property generate item ::=
property_loop_generate_construct
| property_conditional_generate_construct
| property_declaration
| sequence_item

property_loop_generate_construct ::=
for (genvar_initialization **;** genvar_expression **;** genvar_iteration)
property_generate_block

property_conditional_generate_construct ::=
property_if_generate_construct
| property_case_generate_construct

property_if_generate_construct ::=
if (constant_expression **)** property_generate_block_or_null [**else**
property_generate_block_or_null]

property_case_generate_construct ::=
case (constant_expression) property_case_generate_item { case_generate_item } **endcase**

property_case_generate_item ::=
constant_expression { **,** constant_expression } **:** property_generate_block_or_null
| **default** [**:**] property_generate_block_or_null

property_generate_block_or_null ::= property_generate_block | **;**

property_generate_block ::=
property_generate_item
| [generate_block_identifier **:**] **begin** [**:** generate_block_identifier]
{ property_generate_item }
end [**:** generate_block_identifier]

INSERT

17.11.4 Generate constructs in property declaration

Note to editor: Shift the numeration of the following subsections accordingly.

Generate constructs provide the ability for actual arguments to affect the property implementation, thus making properties more flexible and generic. Since generate schemes are evaluated during the model elaboration all their control expressions should be known at the elaboration time.

Consider the following example.

```
sequence s(x , n);
   x[*n];
endsequence

// Assumed n > 0
property p(a, b, m, n);
   generate
     if (m >= 0) begin : prop
        property a_b;
           a |-> b[*m : n];
        endproperty
     end : prop
```

```
      else begin : prop
        property a_b;
          b |-> s(a, m).ended ##0 a[*0 : n];
        endproperty
      end : prop
    endgenerate
  prop.a_b; // instantiate generated property
endproperty
```

This property states that when a happens, b should be asserted in the the time window [m:n]. If m is negative then b should be asserted starting from –m*th* cycle before the occurrence of a until the n*th* cycle after it.

Generate constructs may be used together with the type query functions (Note to editor: Put a reference here), as shown in the following example:

```
property weak_until(p, q);
  generate
    if ($isintegral(q)) begin : prop
      property p;
        !q [*1: $] |-> p;
      endproperty
    end : prop
    else begin : prop
      property p;
        q or (p and ( 1 'b1 |=> weak_until(p, q)));
      endproperty
    end : prop
  endgenerate
  prop.p; // instantiate generated property
endproperty
```

When the second argument of the property weak_until is boolean, the property may be specified directly, while in the general case the recursive form is required. Direct form is usually more efficient for the simulation, and the case when q is boolean is common in practice. Using generate constructs thus allows a more efficient implementation for special cases, while the implementation details are transparent to the end user.

NOTES:

1. As the syntax indicates, it is required to explicitly specify generate block within the property. Omitting could lead to a less intuitive code.
2. A sequence shall have exactly one sequence_expr upon its generate block instantiation. This sequence_expr shall be the last sequence_item.
3. A property shall have exactly one property_spec upon its generate block instantiation. This property_spec shall be the last property_item.

NOTE Change numbering for Recursive properties from 17.11.4 to 17.11.5 and shift numbering

In Recursive properties

REPLACE

For example:

```
property prop_always(p);
```

```
  p and (1'b1 |=> prop_always(p));
endproperty
```

is a recursive property that says that the formal argument property `p` must hold at every cycle. This example is useful if the ongoing requirement that property `p` hold applies after a complicated triggering condition encoded in sequence `s`:

```
property p1(s,p);
  s |=> prop_always(p);
endproperty
```

WITH

For example:

```
property prop_always(p);
  p and (1'b1 |=> prop_always(p));
endproperty
```

is a recursive property that says that the formal argument property `p` must hold at every cycle. This example is useful if the ongoing requirement that property `p` hold applies after a complicated triggering condition encoded in sequence `s`:

```
property p1(s,p);
  s |=> prop_always(p);
endproperty
```

If `prop_always` is not to be used anywhere outside of `p1` then it is more appropriate to declared `prop_always` as a local property of `p1`, as shown in the following example:

```
property p1(s,p);
  property prop_always;
    p and (1'b1 |=> p);
  endproperty
  s |=> prop_always(p);
endproperty
```

Note that `p` in `property_always` is bound to the formal argument of `p1` because the property `prop_always` is declared within the body of `p1`.