

This proposal requires the following modifications:

- Modify BNF in Appendix A.8.3 to include **\$** as part of the constant_param_expression BNF. This is not included in this document
- Modify Section 20.2 as shown below
- Add a subsection 22.3 to Section 22 as shown below
- Modify Section 17.6 as shown below

20.2 Parameter declaration syntax

INSERT syntax box 20-1

A module or an interface can have parameters, which are set during elaboration and are constant during simulation. They are defined with data types and default values. With SystemVerilog, if no data type is supplied, parameters default to type **logic** of arbitrary size for Verilog-2001 compatibility and interoperability.

SystemVerilog adds the ability for a parameter to also specify a data type, allowing modules or instances to have data whose type is set for each instance.

```

module ma #( parameter p1 = 1; parameter type p2 = shortint; )
    (input logic [p1:0] i, output logic [p1:0] o);
    p2 j = 0; // type of j is set by a parameter, (shortint unless redefined)
    always @(i) begin
        o = i;
        j++;
    end
endmodule

module mb;
    logic [3:0] i,o;
    ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
endmodule

```

Another enhancement is to allow **\$** to be assigned to parameters of integer types. A parameter to which **\$** is assigned shall only be used wherever **\$** can be specified as a literal constant.

For example, **\$** represents unbounded range specification, where the upper index can be any integer.

```

parameter r2 = $;
property inq1(r1,r2);
    @(posedge clk) a ##[r1:r2] b ##1 c | => d;
endproperty
assert inq1(3);
    ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
endmodule

```

To support whether a constant is **\$**, a system function is provided to test whether a constant is a **\$**. The syntax of the system function is

```

$isunbounded(const_expression);

```

\$isunbounded returns true if the **const_expression** is unbounded. Typically, **\$isunbounded** is used as a condition in the generate statement.

The example below illustrates the benefit of using **\$** in writing properties concisely where the range is parameterized. The checker in the example ensures that a bus driven by signal `en` remains 0, i.e, quiet for the specified minimum (`min_quiet`) and maximum (`max_quiet`) quiet time.

Note that function `$isunbounded` is used for checking the validity actual arguments.

```

interface quiet_time_checker( clk, reset_n, en);
  input reset_n;
  input clk;
  input [1:0] en;
  parameter min_quiet = 0;
  parameter max_quiet = 0;

  generate
    if ( max_quiet == 0) begin
      property quiet_time;
        @(posedge clk) reset_n |-> ($countones(en) == 1);
      endproperty
      a1: assert property (quiet_time);
    end
    else begin
      property quiet_time;
        @(posedge clk)
          (reset_n && ($past(en) != 0) && en == 0)
          |->(en == 0)[*min_quiet:max_quiet]
          ##1 ($countones(en) == 1);
      endproperty
      a1: assert property (quiet_time);
    end
    if ((min_quiet == 0) && ($isunbounded(max_quiet)))
      $display(warning_msg);
  endgenerate
endinterface

quiet_time_checker #(0, 0) quiet_never (clk,1,enables);
quiet_time_checker #(2, 4) quiet_in_window (clk,1,enables);
quiet_time_checker #(0, $) quiet_any (clk,1,enables);

```

Another example below illustrates that by testing for **\$**, a property can be configured according to the requirements. When parameter `max_cks` is unbounded, it is not required to test for `expr` to become false.

```

interface width_checker(clk, reset_n, expr);
  input clk;
  input reset_n;
  input expr;

  parameter min_cks = 1;
  parameter max_cks = 1;

  generate begin
    if ($isunbounded(max_cks)) begin
      property width;
        @(posedge clk)
          (reset_n && $rose(expr)) |-> (expr [* min_cks]);
      endproperty
      a2: assert property (width);
    end
  endgenerate

```

```

    end
    else begin
        property assert_width_p;
            @(posedge clk)
                (reset_n && $rose(expr)) |-> (expr[* min_cks:max_cks])
                    ##1 (!expr);
        endproperty
        a2: assert property (width);
    end
endgenerate
endinterface

width_checker #(3, $) max_width_unspecified (clk,1,enables);
width_checker #(2, 4) width_specified (clk,1,enables);

```

22.3 Range system function

range_function ::= // not in Annex A

\$isunbounded (constant_expression)

The \$isunbounded system function returns true if the argument is \$. Given the declaration:

```
parameter int foo = $;
```

Then \$isunbounded shall return true.

17.6 Declaring sequences

A **sequence** can be declared in

- a module as a *module_or_generate_item*
- an interface as an *interface_or_generate_item*
- a program as a *non_port_program_item*
- a clocking domain as a *clocking_item*
- \$root

Sequences are declared using the following syntax.:

INSERT and ADD the following to Syntax box 17-4

```

actual_arg_expr ::=
    event_expression
    | $

```

Formal arguments can be optionally specified. A formal argument is untyped, and is used for syntactic replacement of a name or an expression in the sequence.

An actual argument can replace an:

- identifier
- expression
- event control expression
- upper range as \$

Note that 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.

```

sequence s1;
  @(posedge clk) a ##1 b ##1 c;
endsequence
sequence s2;
  @(posedge clk) d ##1 e ##1 f;
endsequence
sequence s3;
  @(negedge clk) g ##1 h ##1 i;
endsequence

```

In this example, sequences `s1` and `s2` are evaluated on each successive posedge of `clk`. The sequence `s3` is evaluated on the negedge of `clk`.

Another example of sequence declaration with arguments is shown below:

```

sequence s20_1(data, en);
  (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
endsequence

```

Sequence `s20_1` does not specify a clock. In this case, a clock would be inherited from some external source, such as a **property** or an **assert** statement. A sequence can be referred to by its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. A sequence can be referenced in a **property**, an **assert** statement, or a **cover** statement.

To use **sequence** as a sub-expression or a part of the expression, simply reference its name. The evaluation of a sequence expression that references a sequence is performed the same way as if the sequence expression contained in the **sequence** was a lexical part of the expression, with the formal arguments substituted by the actual ones and the remaining variables that were not arguments substituted from the scope of declaration. An example is shown below:

```

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

```

Sequence `rule` in the preceding example is equivalent to:

```

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

```

Any form of syntactic cyclic dependency of the sequence names is disallowed. The example below illustrates an illegal dependency of `s1` on `s2` and `s2` on `s1`, because it creates a cyclic dependency.

```

sequence s1;
  @(posedge sysclk) (x ##1 s2);
endsequence
sequence s2;
  @(posedge sysclk) (y ##1 s1);
endsequence

```