Correction: Remove the last limitation of Section 12.4.4 (Distribution)
(The correct limitation is listed at the end of Section 12.4) – BNF is also modified

Limitations:

— A **`dist`** operation shall not be applied to **`randc`** variables.

— A **`dist`** expression requires that expression contain at least one **`rand`** variable.

~~— A **`dist`** expression can only be a top-level constraint (not a predicated constraint).~~

Updated Constraints BNF – Replace in section 12.4 and Annex A.1.9

```
constraint_declaration ::=                                          // from Annex A.1.9
            [ static ] constraint constraint_identifier { { constraint_block } }

constraint_block ::=
              solve [ priority ] identifier_list before identifier_list ;
            | constraint_expression

constraint_expression ::=
              expression ;
            | expression => constraint_set
            | if ( expression ) constraint_set [ else constraint_set ]
            | expression dist { dist_list } ;
            | foreach ( array_identifier [ loop_variables ] ) constraint_set

constraint_set ::=
              constraint_expression
            | { { constraint_expression } }

dist_list ::= dist_item { , dist_item }

dist_item ::=
              value_range := expression
            | value_range :/ expression

constraint_prototype ::= [ static ] constraint constraint_identifier

extern_constraint_declaration ::=
            [ static ] constraint class_identifier :: constraint_identifier { { constraint_block } }

identifier_list ::= identifier { , identifier }

loop_variables ::= [ index_indentifier ] { ,  [ index_indentifier ] }
```

Modify 12.10.3 as shown -- From $srandom() system task to srandom() method

12.10.3 srandom()

The `srandom()` method allows manually seeding an object's Random Number Generator (RNG). The RNG of a process can be seeded using the `srandom()` method of the process (see Section 9.9).

The prototype of the `srandom()` method is:

```
function void srandom( int seed );
```

The `srandom()` method initializes an object's random number generator using the value of the given seed.

Insert as new 12.10.4 & 5 sub-sections

12.10.4 get_randstate()

The `get_randstate()` method retrieves the current state an object's Random Number Generator (RNG). The state of the RNG associated with a process is retrieved using the `get_randstate()`method of the process (see Section 9.9).

The prototype of the `get_randstate()` method is:

```
function string get_randstate();
```

The get_randstate() function returns a copy of the internal state of the RNG associated with the given object.

The RNG state is a string of unspecified length and format. The length and contents of the string are implementation dependent.

12.10.5 set_randstate()

The `set_randstate()`method sets the state of an object's Random Number Generator (RNG). The state of the RNG associated with a process is set using the `set_randstate()`method of the process (see Section 9.9).

The prototype of the `set_randstate()` method is:

```
function void set_randstate( string state );
```

The set_randstate() function copies the given state into the internal state of an object's RNG.

The RNG state is a string of unspecified length and format. Calling set_randstate() with a string value that was not obtained from get_randstate() — or from a different implementation of get_randstate() — is undefined.

Insert as Section 12.4.10

12.4.10 Functions in Constraints

Some properties are unwieldy or impossible to express in a single expression. For example, the natural way to compute the number of 1's in a packed array uses a loop:

```
function int count_ones ( bit [9:0] w );
    for( count_ones = 0; w != 0; w = w >> 1 )
        count_ones += w & 1'b1;
 endfunction
```

Such a function could be used to constrain other random variables to the number of 1 bits:

```
constraint C1 { length == count_ones( v ) };
```

Without the ability to call a function, this constraint requires the loop to be unrolled and expressed as a sum of the individual bits:

```
constraint C2
{
    length == ((w>>9)&1) + ((w>>8)&1) + ((w>>7)&1) + ((w>>6)&1) + ((w>>5)&1) +
              ((w>>4)&1) + ((w>>3)&1) + ((w>>2)&1) + ((w>>1)&1) + ((w>>0)&1);
}
```

Unlike the count_ones function, more complex properties, which require temporary state or unbounded loops, may be impossible to convert into a single expression. The ability to call functions, thus, enhances the expressive power of the constraint language and reduces the likelihood of errors. Note that the two constraints above are not completely equivalent; C2 is bidirectional (length may constraint w and vice-versa), whereas C1 is not.

To handle these common cases, SystemVerilog allows constraint expressions to include function calls, but it imposes certain semantic restrictions.

— Functions that appear in constraint expressions may not contain output or **ref** arguments (**const ref** are allowed).

— Functions that appear in constraint expressions should be automatic (or preserve no state information) and have no side effects.

— Function that appear in constraints may not modify the constraints, for example, calling rand_mode or constraint_mode methods.

— Functions shall be called before constraints are solved, and their return values shall be treated as state variables.

— Random variables used as function arguments shall establish an implicit variable ordering or priority. Constraints that include only variables with higher priority are solved before other, lower priority, constraints. Random variables solved as part of a higher priority set of constraints become state variables to the remaining set of constraints. For example:

```
class B;
    rand int x, y;
    constraint C { x <= F(y); };
    constraint D { y inside { 2, 4, 8 } };
endclass
```

Forces `y` to be solved before x. Thus, constraint D is solved separately before constraint C, which uses the values of `y` and `F(y)` as a state variables.
Within each prioritized set of constrained, cyclical (**randc**) variables are solved first.

— Circular dependencies created by the implicit variable ordering shall result in an error.

— Function calls in active constraints are executed an unspecified number of times (at least once), in an unspecified order.

And change Section 12.4 to:

The declarative nature of constraints imposes the following restrictions on constraint expressions:

— Functions are allowed with certain limitations (see Section 12.4.10).

— Operators with side effects, such as **++** and **--** are not allowed.

— **randc** variables cannot be specified in ordering constraints (see **solve**...**before** in Section 12.4.8).

— **dist** expressions cannot appear in other expressions ~~(unlike **inside**); they can only be top-level expressions~~.

Insert as new 12.4.7 section (After if..else constraints)

12.4.7 Iterative Constraints

Iterative constraints allow arrayed variables to be constrained in a parameterized manner using loop variables and indexing expressions.

The syntax to define an iterative constraint is:

```
constraint_expression ::=                                    // from Annex A.1.9
        …
        | foreach (array_identifier [ loop_variables ] )  constraint_set

loop_variables ::= [ index_indentifier ]{ ,  [ index_indentifier ] }
```

The **foreach** construct specifies iteration over the elements of an array. Its argument is an identifier that designates any type of array (fixed-size, dynamic, or associative) followed by a list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array.

For example:

```
class C;
    rand byte A[] ;

    constraint C1 { foreach ( A [ i ] ) A[i] inside {2,4,8,16}; }
    constraint C2 { foreach ( A [ j ] ) A[j] > 2 * j; }
 endclass
```

C1 constrains each element of the array *A* to be in the set `[2,4,8,16]`. C2 constrains each element of the array *A* to be greater than twice its index.

The number of loop variables must not exceed the number of dimensions of the array variable. The scope of each loop variable is the **foreach** constraint construct, including its constraint_set. The type of each loop variable is implicitly declared to be consistent with the type of array index. An empty loop variable indicates no iteration over that dimension of the array. As with default arguments, a list of commas at the end may be omitted, thus, **foreach**(arr [j]) is a shorthand for **foreach**(arr [j, , , ,]). It shall be an error for any loop variable to have the same identifier as the array.

The mapping of loop variables to array indexes is determined by the dimension cardinality, as described in Section 22.4.

```
//      1  2  3             3    4        1   2     -> Dimension numbers
int A [2][3][4];     bit [3:0][2:1] B [5:1][4];

foreach( A [ i, j, k ] ) …
foreach( B [ q, r, , s ] ) …
```

The first **foreach** causes `i` to iterate from 0 to 1, `j` from 0 to 2, and `k` from 0 to 3. The second **foreach** causes `q` to iterate from 5 to 1, `r` from 0 to 3, and `s` from 2 to 1.

Iterative constraints may include predicates. For example:

```
class C;
    rand int A[] ;

    constraint c1 { arr.size inside {1:10}; }
```

```
    constraint c2 { foreach ( A[ k ] ) (k < A.size - 1) =>  A[k + 1] > A[k]; }
endclass
```

The first constraint, c1, constrains the size of the array A to be between 1 and 10. The second constraint, c2, constrains each array value to be greater than the preceding one, i.e., an array sorted in ascending order.

Within a **foreach**, predicate expressions involving only constants, state variables, object handle comparisons, loop variables, or the size of the array being iterated behave as guards against the creation of constraints, and not as logical relations. For example, the implication in constraint c2 above involves only a loop variable and the size of the array being iterated, thus, it prevents the creation of a constraint when k < A.size() – 1, which in this case prevents an out of bounds access in the constraint. Guards are described in more detail in Section 12.4.11

Index expressions may include loop variables, constants, and state variables. Invalid or out or bound array indexes are not automatically eliminated; users must explicitly exclude these indexes using predicates.

The size method of a dynamic or associative array can be used to constrain the size of the array (see constraint c1 above). If an array is constrained by both size constraints and iterative constraints, the size constraints are solved first, and the iterative constraints next. As a result of this implicit ordering between size constraints and iterative constraints, the size method shall be treated as a state variable within the **foreach** block of the corresponding array. For example, the expression A.size is treated as a random variable in constraint c1, and as a state variable in constraint c2. This implicit ordering can cause the solver to fail in some situations.

Insert as Section 12.4.11

12.4.11 Constraint guards

Constraint guards are predicate expressions that function as guards against the creation of constraints, and not as logical relations to be satisfied by the solver. These predicate expressions are evaluated before the constraints are solved, and are characterized by involving only the following items:

- constants
- state variables
- object handle comparisons (comparisons between two handles or a handle and the constant **null**)

In addition to the above, iterative constraints (see Section 12.4.7) also consider loop variables and the size of the array being iterated as state variables.

Treating these predicate expressions as constraint guards prevents the solver from generating evaluation errors, thereby failing on some seemingly correct constraints. This enables users to write constraints that avoid errors due to nonexistent object handles or array indices out of bounds. For example, the `sort` constraint of the singly-linked list, `SList`, shown below is intended to assign a random sequence of numbers that is sorted in ascending order. However, the constraint expression will fail on the last element when `next.n` results in an evaluation error due to a non-existent handle.

```
class SList;
    rand int n;
    rand Slist next;

    constraint sort  {  n < next.n; }
endclass
```

The error condition above can be avoided by writing a predicate expression to guard against that condition:

```
constraint sort { if( next != null ) n < next.n; }
```

In the `sort` constraint above, the **`if`** prevents the creation of a constraint when next == **null**, which in this case avoids accessing a non-existent object. Both implication (=>) and **if…else** can be used as guards.

Guard expressions can themselves include sub-expressions that result in evaluation errors (e.g., null references), and they are also guarded from generating errors. This logical sifting is accomplished by evaluating predicate sub-expressions using the following 4-state representation:

- **0** **TRUE** Sub-expression evaluates to TRUE
- **1** **FALSE** Sub-expression evaluates to FALSE
- **E** **ERROR** Sub-expression causes an evaluation error
- **R** **RANDOM** Expression includes random variables and cannot be evaluated

Every sub-expression within a predicate expression is evaluated to yield one of the above four values. The sub-expressions are evaluated in an arbitrary order, and the result of that evaluation plus the logical operation define the outcome in the alternate 4-state representation. A conjunction (**`&&`**), disjunction (**`||`**), or negation (**`!`**) of sub-expressions can include some (perhaps all) guard sub-expressions. The following rules specify the resulting value for the guard:

- **Conjunction (`&&`):** If any one of the sub-expressions evaluates to FALSE then the guard evaluates to FALSE. Otherwise, if any one sub-expression evaluates to ERROR then the guard evaluates to ERROR, else the guard evaluates to TRUE.
  - If the guard evaluates to FALSE then the constraint is eliminated.
  - If the guard evaluates to TRUE then a (possibly conditional) constraint is generated.
  - If the guard evaluates to ERROR then an error is generated and randomize fails.

— **Disjunction (`||`):** If any one of the sub-expressions evaluates to TRUE then the guard evaluates to TRUE. Otherwise, if any one sub-expression evaluates to ERROR then the guard evaluates to ERROR, else the guard evaluates to FALSE.
  — If the guard evaluates to FALSE then a (possibly conditional) constraint is generated.
  — If the guard evaluates to TRUE then an unguarded constraint is generated.
  — If the guard evaluates to ERROR then an error is generated and randomize fails.
  —
— **Negation (`!`):** If the sub-expression evaluates ERROR then the guard evaluates to ERROR. Otherwise, if the sub-expression evaluates to TRUE or FALSE then the guard evaluates to FALSE or TRUE, respectively.

These rules are codified by the following truth tables:

&&	0	1	E	R
0	0	0	0	0
1	0	1	E	R
E	0	E	E	E
R	0	R	E	R

Conjunction

||	0	1	E	R
0	0	1	E	R
1	1	1	1	1
E	E	1	E	E
R	R	R	E	R

Disjunction

!	
0	1
1	0
E	E
R	R

Negation

These above rules are applied recursively until all sub-expressions are evaluated. The final value of the evaluated predicate expression determines the outcome as follows:
— If the result is TRUE then an unconditional constraint is generated.
— If the result is FALSE then the constraint is eliminated, and can generate no error.
— If the result is ERROR then an unconditional error is generated and the constraint fails.
— If the final result of the evaluation is RANDOM then a guarded constraint is generated.

When final value is RANDOM a traversal of the predicate expression tree is needed to collect all conditional guards that evaluate to RANDOM. When the final value is ERROR, a subsequent traversal of the expression tree is not required, allowing implementations to issue only one error.

Example 1:

```
class D;
    int x;
endclass

class C;
    rand int x, y;
    D a,b;
    constraint c1 { (x < y || a.x > b.x || a.x == 5 ) => x+y == 10; }
endclass
```

In the example above, the sub-expressions of the guard are `(x < y)`, `(a.x > b.x)`, and `(a.x == 5)`, which are all connected by disjunction. We consider several cases:

Case 1: `a` is `non-null`, `b` is **`null`**, `a.x` is `5`.
Since (`a.x==5`) is true, the fact that `b.x` generates an error does not result in an error.
The unconditional constraint (`x+y == 10`) is generated.

Case 2: `a` is **`null`**
This always results in error, irrespective of the other conditions.

Case 3: `a` is `non-null`, `b` is `non-null`, `a.x` is 10, `b.x` is 20.
All the guard sub-expressions evaluate to FALSE, producing constraint `(x<y) => (x+y == 10)`.

Example 2:

```
class D;
    int x;
endclass
```

```
class C;
    rand int x, y;
    D a,b;
    constraint c1 { (x < y && a.x > b.x && a.x == 5 ) => x+y == 10; }
endclass
```

In the example above, the sub-expressions of the guard are `(x < y)`, `(a.x > b.x)`, and `(a.x == 5)`, which are all connected by conjunction. We consider the following cases:

Case 1: `a` is `non-`**`null`**`, b` is **`null`**`, a.x` is `6.`
Since (`a.x==5`) is false, the fact that `b.x` generates an error does not result in an error.
The constraint is eliminated.

Case 2: `a` is **`null`**
This always results in error, irrespective of the other conditions.

Case 3: `a` is `non-`**`null`**`, b` is `non-`**`null`**, `a.x` is 5, `b.x` is 2.
All the guard sub-expressions evaluate to TRUE, producing constraint `(x<y) => (x+y == 10).`

Example 3:

```
class D;
    int x;
endclass

class C;
    rand int x, y;
    D a,b;
    constraint c1 { (x < y && (a.x > b.x || a.x ==5)) => x+y == 10; }
endclass
```

In the example above, the sub-expressions of the guard are `(x < y)` and `(a.x > b.x || a.x == 5)`, which are connected by disjunction. We consider the following cases:

Case 1: `a` is `non-`**`null`**`, b` is **`null`**`, a.x` is `5.`
The guard expression evaluates to (ERROR || `a.x==5`), which evaluates to (ERROR || TRUE).
The guard sub-expression evaluates to TRUE, producing constraint `(x<y) => (x+y == 10).`

Case 2: `a` is `non-`**`null`**`, b` is **`null`**`, a.x` is `8.`
The guard expression evaluates to (ERROR || FALSE), and generates an error.

Case 3: `a` is **`null`**
This always results in error, irrespective of the other conditions.

Case 4: `a` is `non-`**`null`**`, b` is `non-`**`null`**, `a.x` is 5, `b.x` is 2.
All the guard sub-expressions evaluate to TRUE, producing constraint `(x<y) => (x+y == 10).`

Insert as new 12.10 section – Before Section "Random number system functions"

12.10 In-line random variable control

The `randomize()` method can be used to temporarily control the set of random and state variables within a class instance or object. When the randomize method is called with no arguments, it behaves as described in the previous sections, that is, it assigns new values to all random variables in an object --- those declared as **rand** or **randc** --- such that all of the constraints are satisfied. When randomize is called with arguments, those arguments designate the complete set of random variables within that object; all other variables in the object are considered state variables. For example, consider the following class and calls to randomize:

```
class CA;
    rand byte x, y;
    byte v, w;

    constraint c1 { x < v && y > w );
endclass

CA a = new;

a.randomize();          // random variables: x, y   state variables: v, w
a.randomize( x );       // random variables: x      state variables: y, v, w
a.randomize( v, w );    // random variables: v, w   state variables: x, y
a.randomize( w, x );    // random variables: w, x   state variables: y, v
```

This mechanism controls the set of active random variables for the duration of the call to randomize, which is conceptually equivalent to making a set of calls to the rand_mode() method to disable or enable the corresponding random variables. Calling randomize() with arguments allows changing the random mode of any class property, even those not declared as **rand** or **randc**. This mechanism, however, does not affect the cyclical random mode: it cannot change a non-random variable into a cyclical random variable (**randc**), and cannot change a cyclical random variable into a non-cyclical random variables (change from **randc** to **rand**).

The scope of the arguments to the randomize method is the object class. Arguments are limited to the names of properties of the calling object; expressions are not allowed. The random mode of **local** class members can only be changed when the call to randomize has access to those properties, that is, within the scope of the class in which the local members are declared.

12.10.1 In-line constraint checker

Normally, calling the randomize method of a class that has no random variables causes the method to behave as a checker, that is, it assigns no random values, and only returns a status: one if all constraints are satisfied and zero otherwise. The in-line random variable control mechanism can also be used to force the randomize() method to behave as a checker.

The randomize method accepts the special argument **null** to indicate no random variables for the duration of the call, that is, all class members behave as state variables. This causes the randomize method to behave as a checker instead of a generator. A checker evaluates all constraints and simply returns one if all constraints are satisfied, and zero otherwise. For example, if class CA defined above executes the following call:

```
success = a.randomize( null );      // no random variables
```

Then the solver considers all variables as state variables and only checks whether the constraint is satisfied, namely, that the relation `(x < v && y > w)` is true using the current values of `x, y, v,` and `w`.

12.11 Randomization of scope variables - ::randomize()

The built-in class randomize method operates exclusively on class member variables. Using classes to model the data to be randomized is a powerful mechanism that enables the creation of generic, reusable objects containing random variables and constraints that can be later extended, inherited, constrained, overridden, enabled, disabled, merged with or separated from other objects. The ease with which classes and their associated random variables and constraints can be manipulated make classes an ideal vehicle for describing and manipulating random data and constraints. However, some less-demanding problems that do not require the full flexibility of classes, can use a simpler mechanism to randomize data that does not belong to a class. The scope randomize method, `::randomize()`, enables users to randomize data in the current scope, without the need to define a class or instantiate a class object.

The syntax of the scope randomize method is:

```
scope_randomize ::=                                                    // from Annex A.?
        [::] randomize ( [ variable_identifier_list ] ) [ with {  constraint_block  } ]

variable_identifier_list ::= variable_identifier {, variable_identifier }
```

The scope randomize method behaves exactly the same as a class randomize method, except that it operates on the variables of the current scope instead of class member variables. Arguments to this method specify the variables that are to be assigned random values. i.e., the random variables.

For example:

```
module stim;
    bit[15:0] addr;
    bit[31:0] data;

    function bit gen_stim();
        bit success, rd_wr;

        success = ::randomize( addr, data, rd_wr );
        return rd_wr ;
    endfunction

    ...
endmodule
```

The function *gen_stim* calls `::randomize()` with three variables as arguments: *addr, data,* and *rd_wr*. Thus, ::randomize() assigns new random variables to those variables that are visible in the scope of the *gen_stim* function. Note that *addr* and *data* have module scope, whereas *rd_wr* has scope local to the function. The above example can also be written using a class:

```
class stimc;
    rand bit[15:0] addr;
    rand bit[31:0] data;
    rand bit rd_wr;
endclass

function bit gen_stim( stimc p );
    bit success;
    success = p.randomize();
    addr = p.addr;
    data = p.data;
    return p.rd_wr;
endfunction
```

However, for this simple application, the scope randomize method leads to a straightforward implementation.

The scope randomize method returns 1 if it successfully sets all the random variables to valid values, otherwise it returns 0. If the scope randomize method is called with no arguments then it behaves as a checker, and simply returns status.

12.11.1 Adding constraints to scope variables - ::randomize() with

The `::randomize()` **`with`** form of the scope randomize method allows users to specify random constraints to be applied to the local scope variables. When specifying constraints, the arguments to the scope randomize method become random variables, all other variables are considered state variables.

```
task stimulus( int length );
    int a, b, c, success;

    success = ::randomize( a, b, c ) with { a < b ; a + b < length };
    ...
    success = ::randomize( a, b ) with { b - a > length };
    ...
endfunction
```

The task stimulus above calls ::randomize twice resulting in two sets of random values for its local variables a, b, and c. In the first call variables a and b are constrained such that variable a is less than b, and their sum is less that the task argument length, which is designated as a state variable. In the second call, variables a and b are constrained such that their difference is greater than state variable length.

Add to the class process prototype in Section 9.9 the text in blue

```
class process;
    enum state { FINISHED, RUNNING, WAITING, SUSPENDED, KILLED };

    static function process self();
    function state status();
    task kill();
    task await();
    task suspend();
    task resume();

    function void srandom( int seed );
    function string get_randstate();
    function void set_randstate( string state );
endclass
```

Add at the end of Section 9.9 the following text

The **srandom()** function allows manually seeding the Random Number Generator (RNG) of a process. The top level RNG of each program is initialized with process::self.srandom(1) prior to any randomization calls.

The **get_randstate()** function retrieves the current state of the Random Number Generator associated with the given process. The RNG state is a string of unspecified length and format (see Section 12.10.4).

The **set_randstate()** sets the state of the Random Number Generator (RNG) associated with the given process to the specified string (see Section 12.10.5).