SystemVerilog 3.1
Random Constraints - Proposal

Version 1.0
November 18, 2002

Index

SystemVerilog 3.1
Random Constraints

1 Introduction

Constraint-driven test generation allows users to automatically generate tests for functional verification. Random testing can be more effective than a traditional, directed testing approach. By specifying constraints, one can easily create tests that can find hard-to-reach corner cases. This proposal allows users to specify constraints in a compact, declarative way. The constraints are then processed by a solver that generates random values that meet the constraints.

The random constraints are built on top of an object oriented framework that models the data to be randomized as objects that contain random variables and user-defined constraints. The constraints determine the legal values that can be assigned to the random variables. Objects are ideal for representing complex aggregate data types and protocols such as Ethernet packets.

The next section provides an overview of object-based randomization and constraint programming. The rest of this document provides detailed information on random variables, constraint blocks, and the mechanisms used to manipulate them.

2 Overview

This section introduces the basic concepts and uses for generating random stimulus within objects. The proposed language uses an object-oriented method for assigning random values to the member variables of an object subject to user-defined constraints. For example:

```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;

   constraint word_align {addr[1:0] == ‘2b0;}
endclass
```

The Bus class models a simplified bus with two random variables: *addr* and *data*, representing the address and data values on a bus. The word_align constraint declares that the random values for *addr* must be such that *addr* is word-aligned (the low-order 2 bits are 0).

The **randomize()** method is called to generate new random values for a bus object:

```
Bus bus = new;

repeat (50) begin
   if( bus.randomize() == 1 )
      $display ("addr = %16h data = %h\n", bus.addr, bus.data);
   else
      $display ("Randomization failed.\n");
end
```

Calling **randomize**() causes new values to be selected for all of the random variables in an object such that all of the constraints are true ("satisfied"). In the program test above, a "bus" object is created and then randomized 50 times. The result of each randomization is checked for success. If the randomization succeeds, the new random values for *addr* and *data* are printed; if the randomization fails, an error message is printed. In this example, only the *addr* value is constrained, while the *data* value is unconstrained. Unconstrained variables are assigned any value in their declared range.

Constraint programming is a powerful method that lets users build generic, reusable objects that can later be extended or constrained to perform specific functions. The approach differs from both traditional procedural and object-oriented programming, as illustrated in this example that extends the Bus class:

```
typedef enum {low, mid, high} AddrType;

class MyBus extends Bus;
   rand AddrType type;
   constraint addr_range
   {
      (type == low ) => addr inside {   [0 : 15] };
      (type == mid ) => addr inside {  [16 : 127]};
      (type == high) => addr inside {[128 : 255]};
   }
endclass
```

The MyBus class inherits all of the random variables and constraints of the Bus class, and adds a random variable called type that is used to control the address range using another constraint. The *addr_range* constraint uses implication to select one of three range constraints depending on the random value of type. When a MyBus object is randomized, values for *addr, data,* and *type* are computed such that all of the constraints are satisfied. Using inheritance to build layered constraint systems allows uses to develop general-purpose models that can be constrained to perform application-specific functions.

Objects can be further constrained using the **randomize() with** construct, which declares additional constraints inline with the call to **randomize**():

```
task exercise_bus (MyBus bus);
   int res;

      // EXAMPLE 1: restrict to small addresses
   res = bus.randomize() with {type == small;};

      // EXAMPLE 2: restrict to address between 10 and 20
   res = bus.randomize() with {10 <= addr && addr <= 20;};

      // EXAMPLE 3: restrict data values to powers-of-two
   res = bus.randomize() with {data & (data - 1) == 0;};
endtask
```

This example illustrates several important properties of constraints:

— Constraints can be any SystemVerilog expression with variables and constants of integral type (**bit, reg, logic, integer, enum**, **packed struct**, etc…).
— Constraint expressions follow SystemVerilog syntax and semantics, including precedence, associativity, sign extension, truncation, and wrap-around.
— The constraint solver must be able to handle a wide spectrum of equations, such as algebraic factoring, complex Boolean expressions, and mixed integer and bit expressions. In the example above, the power-of-two constraint was expressed arithmetically. It could have also been defined with expressions using a shift operator. For example, 1 << *n*, where *n* is a 5-bit random variable.
— If a solution exists, the constraint solver must find it. The solver may fail only when the problem is over-constrained and there is no combination of random values that satisfy the constraints.
— Constraints interact *bi-directionally*. In this example, the value chosen for *addr* depends on *type* and how it is constrained, and the value chosen for *type* depends on *addr* and how it is constrained. All expression operators are treated bi-directionally, including the implication operator (=>).

Sometimes it is desirable to disable constraints on random variables. For example, consider the case where we want to deliberately generate an illegal address (non-word aligned):

```
task exercise_illegal(MyBus bus, int cycles);
   int res;

      // Disable word alignment constraint.
   $constraint_mode( OFF, bus.word_align );

   repeat (cycles) begin

         // CASE 1: restrict to small addresses.
      res = bus.randomize() with {addr[0] || addr[1];};
      ...
   end

      // Re-enable word alignment constraint.
   $constraint_mode( ON, bus.word_align );
endtask
```

The **$constraint_mode()** system task can be used to enable or disable any named constraint block in an object. In this example, the word-alignment constraint is disabled, and the object is then randomized with additional constraints forcing the low-order address bits to be non-zero (and thus unaligned).

The ability to enable or disable constraints allows users to design a constraint hierarchies. In these hierarchies, the lowest level constraints can represent physical limits grouped by common properties into named constraint blocks, which can be independently enabled or disabled.

Similarly, the **$rand_mode()** method can be used to enable or disable any random variable. When a random variable is disabled, it behaves in exactly the same way as other non-random variables.

Occasionally, it is desirable to perform operations immediately before or after randomization. That is accomplished via two built-in methods, **pre_randomize()** and **post_randomize(),** which are automatically called before and after randomization. These methods can be overloaded with the desired functionality:

```
class XYPair;
   rand integer x, y;
endclass

class MyYXPair extends XYPair
   function void pre_randomize();
      super.pre_randomize();
      printf("Before randomize x=%0d, y=%0d\n", x, y);
   endtask

   function void post_randomize();
      super.post_randomize();
      printf("After randomize x=%0d, y=%0d\n", x, y);
   endtask
endclass
```

By default, **pre_randomize()** and **post_randomize()** call their overloaded parent class methods. When **pre_randomize()** or **post_randomize()** are overloaded, care must be taken to invoke the parent class' methods, unless the class is a base class (has no parent class).

The random stimulus generation capabilities and the object-oriented constraint-based verification methodology enable users to quickly develop tests that cover complex functionality and better assure design correctness.

3 Random Variables

Class variables can be declared random using the **rand** and **randc** type-modifier keywords.

The syntax to declare a random variable in a class is:

rand *variable*;

or

randc *variable*;

— The solver can randomize scalar variables of any integral type such as integer, enumerated types, and packed array variables of any size.

— Arrays can be declared **rand** or **randc**, in which case all of their member elements are treated as **rand** or **randc**.

— Dynamic and associative arrays can be declared **rand** or **randc**. All of the elements in the array are randomized. If the array elements are of type object, all of the array elements must be non-**null**. Individual array elements may be constrained, in which case the index expression must be a literal constant.

— The size of a dynamic array declared as **rand** or **randc** may also be constrained**.** In that case, the array will be resized according to the size constraint, and then all the array elements will be randomized. The array size constraint is declared using the **size** method. For example,

```
rand bit[7:0] len;
rand integer data[*];
constraint db { data.size == len );
```

The variable *len* is declared to be 8 bits wide. The randomizer computes a random value for the *len* variable in the 8-bit range of 0 to 255, and then randomizes the first *len* elements of the data array.

If a dynamic array's **size** is not constrained then **randomize()** randomizes all the elements in the array.

— An object variable can be declared **rand** in which case all of that object's variables and constraints are solved concurrently with the other class variables and constraints. Objects cannot be declared **randc**.

3.1 rand Modifier

Variables declared with the **rand** keyword are standard random variables. Their values are uniformly distributed over their range. For example:

```
rand bit[7:0] y;
```

This is an 8-bit unsigned integer with a range of 0 to 255. If unconstrained, this variable will be assigned any value in the range 0 to 255 with equal probability. In this example, the probability of the same value repeating on successive calls to randomize is 1/256.

3.2 randc Modifier

Variables declared with the **randc** keyword are random-cyclic variables that cycle through all the values in a random permutation of their declared range. Random-cyclic variables can only be of type **bit, char,** or enumerated types, and may be limited to a maximum size.

To understand **randc**, consider a 2-bit random variable y:

```
randc bit[1:0] y;
```

The variable y can take on the values 0, 1, 2, and 3 (range 0 to 3). Randomize computes an initial random permutation of the range values of y, and then returns those values in order on successive calls. After it returns the last element of a permutation, it repeats the process by computing a new random permutation.

The basic idea is that **randc** randomly iterates over all the values in the range and that no value is repeated within an iteration. When the iteration finishes, a new iteration automatically starts.

```
initial permutation:     0 → 3 → 2 → 1 ─┐
                        ┌───────────────┘
 next permutation:      └→ 2 → 1 → 3 → 0 ─┐
                        ┌─────────────────┘
 next permutation:      └→ 2 → 0 → 1 → 3 ...
```

The permutation sequence for any given **randc** variable is recomputed whenever the constraints change on that variable, or when none of the remaining values in the permutation can satisfy the constraints.

To reduce memory requirements, implementations may impose a limit on the maximum size of a **randc** variable, but it should be no less than 8 bits.

The semantics of cyclical variables requires that they be solved before other random variables. A set of constraints that includes both **rand** and **randc** variables will be solved such that the **randc** variables are solved first, and this may sometimes cause **randomize()** to fail.

4 Constraint Blocks

The values of random variables are determined using constraint expressions that are declared using constraint blocks. Constraint blocks are class members, like tasks, functions, and variables. They must be defined after the variable declarations in the class, and before the task and function declarations in the class. Constraint block names must be unique within a class.

The syntax to declare a constraint block is:

constraint *constraint_name* **{** *contraint_expressions* **}**

constraint_name is the name of the constraint block. This name can be used to enable or disable a constraint using the system task **$constraint_mode()**.

constraint_expressions is a list of expression statements that restrict the range of a variable or define relations between variables. A constraint expression is any SystemVerilog expression, or one of the constraint-specific operators: **=>**, **inside** and **dist**.

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

— Calling tasks or functions is not allowed.
— Operators with side effects, such as **++** and **--** are not allowed.
— **randc** variables cannot be specified in ordering constraints (see **solve..before** in Section 12).
— **dist** expressions cannot appear in other expressions (unlike **inside**); they can only be top-level expressions.

5 External Constraint Blocks

Constraint block bodies can be declared outside a class declaration, just like external task and function bodies:

```
// class declaration
class XYPair;
   rand integer x, y;
   constraint c;
endclass

// external constraint body declaration
constraint XYPair::c { x < y; }
```

6 Inheritance

Constraints follow the same general rules for inheritance as class variables, tasks, and functions:

— A constraint in a derived class that uses the same name as a constraint in its parent classes effectively overrides the base class constraints. For example:

```
class A;
   rand integer x;
   constraint c { x < 0; }
endclass

class B extends A;
   constraint c { x > 0; }
endclass
```

An instance of class A constrains x to be less than zero whereas an instance of class B constrains x to be greater than zero. The extended class B overrides the definition of constraint c. In this sense, constraints are treated the same as virtual functions, so casting an instance of B to an A does not change the constraint set.

— The **randomize()** task is virtual, accordingly, it treats the class constraints in a virtual manner. When a named constraint is overloaded, the previous definition is overriden.

— The built-in methods **pre_randomize()** and **post_randomize()** are functions and cannot block.

7 Set Membership

Constraints support integer value sets and set membership operators.

The syntax to define a set expression is:

expression **inside {** *value_range_list* **}**;

or

expression **inside** *array*; // fixed-size, dynamic, or associative array *expression* is any integral SystemVerilog expression.

value_range_list is a comma-separated list of integral expressions and ranges. Ranges are specified with a low and high bound, enclosed by square braces [], and separated by a colon: [*low_bound* : *high_bound*]. Ranges include all of the integer elements between the bounds. The bound to the left of the colon MUST be less than or equal to the bound to the right, otherwise the range is empty and contains no values.

The **inside** operator evaluates to true if the expression is contained in the set; otherwise it evaluates to false.

Absent any other constraints, all values (either single values or value ranges), have an equal probability of being chosen by the **inside** operator.

The negated form denotes that expression lies outside the set: !(*expression* **inside** **{** *set* **}**)

For example:

```
rand integer x, y, z;
constraint c1 {x inside {3, 5, [9:15], [24:32], [y:2*y], z};}

rand integer a, b, c;
constraint c2 {a inside {b, c};}
```

Set values and ranges can be any integral expression. Values can be repeated, so values and value ranges can overlap. It is important to note that the **inside** operator is bidirectional, thus, the second example is equivalent to `a == b || a == c`.

8 Distribution

In addition to set membership, constraints support sets of weighted values called distributions. Distributions have two properties: they are a relational test for set membership, and they specify a statistical distribution function for the results.

The syntax to define a distribution expression is:

expression **dist { *value_range_ratio_list* }**;

expression can be any integral SystemVerilog expression.

The distribution operator **dist** evaluates to true if the expression is contained in the set; otherwise it evaluates to false.

Absent any other constraints, the probability that the expression matches any value in the list is proportional to its specified weight.

The *value_range_ratio_list* is a comma-separated list of integral expressions and ranges (the same as the *value_range_list* for set membership). Optionally, each term in the list can have a weight, which is specified using the `:=` or `:/` operators. If no weight is specified, the default weight is 1. The weight may be any integral SystemVerilog expression.

The `:=` operator assigns the specified weight to the item, or if the item is a range, to every value in the range.

The `:/` operator assigns the specified weight to the item, or if the item is a range, to the range as a whole. If there are *n* values in the range, the weight of each value is *range_weight* / *n*.

For example:

```
x dist {100 := 1, 200 := 2, 300 := 5}
```

means x is equal to 100, 200, or 300 with weighted ratio of 1-2-5. If an additional constraint is added that specifies that x cannot be 200:

```
x != 200;
x dist {100 := 1, 200 := 2, 300 := 5}
```

then x is equal to 100 or 300 with weighted ratio of 1-5.

It is easier to think about mixing ratios, such as 1-2-5, than the actual probabilities because mixing ratios do not have to be normalized to 100%. Converting probabilities to mixing ratios is straightforward.

When weights are applied to ranges, they can be applied to each value in the range, or they can be applied to the range as a whole. For example,

```
x dist { [100:102] := 1, 200 := 2, 300 := 5}
```

means x is equal to 100, 101, 102, 200, or 300 with a weighted ratio of 1-1-1-2-5.

```
x dist { [100:102] :/ 1, 200 := 2, 300 := 5}
```

means x is equal to one of 100, 101, 102, 200, or 300 with a weighted ratio of 1/3-1/3-1/3-2-5.

In general, distributions guarantee two properties: set membership and monotonic weighting, which means that increasing a weight will increase the likelihood of choosing those values.

Limitations:
— A **dist** operation may not be applied to **randc** variables.
— A **dist** expression requires that expression contain at least one **rand** variable.

9 Implication

Constraints provide two constructs for declaring conditional (predicated) relations: implication and if-else.

The implication operator (**=>**) can be used to declare an expression that implies a constraint.

The syntax to define an implication constraint is:

expression **=>** constraint;
expression **=>** constraint_block;

The *expression* can be any integral SystemVerilog expression.

The implication operator **=>** evaluates to true if the expression is false or the constraint is satisfied; otherwise it evaluates to false.

The *constraint* is any valid constraint, and *constraint_block* represents an anonymous constraint block. If the expression is true, all of the constraints in the constraint block must also be satisfied.

For example:

```
mode == small => len < 10;
mode == large => len > 100;
```

In this example, the value of *mode* implies that the value of *len* is less than 10 or greater than 100. If *mode* is neither small nor large, the value of *len* is unconstrained.

The boolean equivalent of (a **=>** b) is (!a || b). Implication is a bidirectional operator. Consider the following example:

```
bit[3:0] a, b;
constraint c {(a == 0) => (b == 1)};
```

Both a and b are 4 bits, so there are 256 combinations of a and b. Constraint c says that a == 0 implies that b == 1, thereby eliminating 15 combinations: {0,0}, {0,2}, … {0,15}. The probability that a == 0 is thus 1/(256-15) or 1/241.

It is important to that the constraint solver be designed to cover the whole random value space with uniform probability. This allows randomization to better explore the whole design space than in cases where certain value combinations are preferred over others.

10 if-else Constraints

If-else style constraint are also supported.

The syntax to define an **if-else** constraint is:

if (*expression*) *constraint*; [**else** *constraint*;]
if (*expression*) *constraint_block* [**else** *constraint_block*]

expression can be any integral SystemVerilog expression.

constraint is any valid constraint. If the expression is true, the first constraint must be satisfied; otherwise the optional else-constraint must be satisfied.

constraint_block represents an anonymous constraint block. If the expression is true, all of the constraints in the first constraint block must be satisfied, otherwise all of the constraints in the optional else-constraint-block must be satisfied. Constraint blocks may be used to group multiple constraints.

If-else style constraint declarations are equivalent to implications:

```
if (mode == small)
    len < 10;
else if (mode == large)
    len > 100;
```

is equivalent to

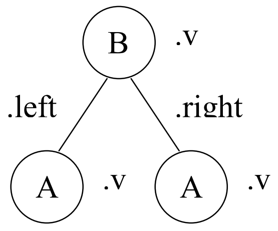
```
mode == small => len < 10 ;
mode == large => len > 100 ;
```

In this example, the value of mode implies that the value of *len* is less than 10, greater than 100, or unconstrained.

Just like implication, if-else style constraints are bi-directional. In the declaration above, the value of *mode* constraints the value of *len*, and the value of *len* constrains the value of *mode*.

11 Global Constraints

When an object member of a class is declared **rand**, all of its constraints and random variables are randomized simultaneously along with the other class variables and constraints. Constraint expressions involving random variables from other objects are called *global constraints.*



```
class A;              // leaf node
    rand bit[7:0] v;
endclass

class B extends A;    // heap node
    rand A left;
    rand A right;

    constraint heapcond {left.v <= v; right.v <= v;}
endclass
```

This example uses global constraints to define the legal values of an ordered binary tree. Class A represents a leaf node with an 8-bit value x. Class B extends class A and represents a heap-node with value v, a left sub-tree, and a right sub-tree. Both sub-trees are declared as **rand** in order to randomize them at the same time as other class variables. The constraint block named *heapcond* has two global constraints relating the left and right sub-tree values to the heap-node value. When an instance of class B is randomized, the solver simultaneously solves for B and its left and right children, which in turn may be leaf nodes or more heap-nodes.

The following rules determines which objects, variables, and constraints are to be randomized:

1. First, determine the set of objects that are to be randomized as a whole. Starting with the object that invoked the **randomize()** method, add all objects that are contained within it, are declared **rand**, and are active (see **$rand_mode**). The definition is recursive and includes all of the active random objects that can be reached from the starting object. The objects selected in this step are referred to as the *active random objects*.

2. Next, select all of the active constraints from the set of active random objects. These are the constraints that are applied to the problem.

3. Finally, select all of the active random variables from the set of active random objects. These are the variables that are to be randomized. All other variable references are treated as state variables, whose current value is used as a constant.

12 Variable Ordering

The solver assures that the random values are selected to give a uniform value distribution over legal value combinations (that is, all combinations of values have the same probability of being chosen). This important property guarantees that all value combinations are equally probable.

Sometimes, however, it is desirable to force certain combinations to occur more frequently. Consider this case where a 1-bit "control" variable *s* constrains a 32-bit "data" value *d:*

```
class B;
   rand bit s;
   rand bit[31:0] d;

   constraint c { s => d == 0; }
endclass
```

The constraint c says "*s* implies *d* equals zero". Although this reads as if *s* determines *d*, in fact *s* and *d* are determined together. There are 2^{32} valid combinations of {s,d}, but *s* is only true for {1,0}. Thus, the probability that *s* is true is $1/2^{32}$, which is practically zero.

The constraints provide a mechanism for order variables so that *s* can be chosen independent of *d*. This mechanism defines a partial ordering on the evaluation of variables, and is specified using the **solve** keyword.

```
class B;
   rand bit s;
   rand bit[31:0] d;
   constraint c { s => d == 0; }
   constraint order { solve s before d; }
endclass
```

In this case, the order constraint instructs the solver to solve for *s* before solving for *d*. The effect is that *s* is now chosen true with 50% probability, and then *d* is chosen subject to the value of *s*. Accordingly, d == 0 will occur 50% of the time, and d != 0 will occur for the other 50%.

Variable ordering can be used to force selected corner cases to occur more frequently than they would otherwise.

The syntax to define variable order in a constraint block is:

solve *variable_list* **before** *variable_list* **;**

variable_list is a comma-separated list of integral scalar variables or array elements.

The following restrictions apply to variable ordering:
— Only random variables are allowed, that is, they must be **rand.**
— **randc** variables are not allowed. **randc** variables are always solved before any other.
— The variables must be integral scalar values.
— A constraint block may contain both regular value constraints and ordering constraints.
— There must be no circular dependencies in the ordering, such as “solve a before b” combined with “solve b before a”.
— Variables that are not explicitly ordered will be solved with the last set of ordered variables. These values are deferred until as late as possible to assure a good distribution of value.
— Variables can be solved in an order that is not consistent with the ordering constraints, provided that the outcome is the same. An example situation where this might occur is:

```
x == 0;
x < y;
solve y before x;
```

In this case, since x has only one possible assignment (0), x can be solved for before y. The constraint solver can use this flexibility to speed up the solving process.

13 Randomization Methods

13.1 randomize()

Variables in an object are randomized using the **randomize()** class method. Every class has a built-in **randomize()** virtual method, declared as:

virtual function int randomize();

The **randomize()** method is a virtual function that generates random values for all the active random variables in the object, subject to the active constraints.

The **randomize()** method returns 1 if it successfully sets all the random variables and objects to valid values, otherwise it returns 0.

Example:

```
class SimpleSum;
   rand bit[7:0] x, y, z;
   constraint c {z == x + y;}
endclass
```

This class definition declares three random variables, x, y, and z. Calling the **randomize()** method will randomize an instance of class SimpleSum:

```
SimpleSum p = new;
int success = p.randomize();
if (success == 1 ) ...
```

Checking results is always needed because the actual value of state variables or addition of constraints in derived classes may render seemingly simple constraints unsatisfiable.

13.2 pre_randomize() and post_randomize()

Every class contains built-in **pre_randomize()** and **post_randomize()** functions, that are automatically called by **randomize()** before and after computing new random values.

Built-in definition for **pre_randomize()**:

```
function void pre_randomize;
   if (super) super.pre_randomize();
   // Optional programming before randomization goes here.
endfunction
```

Built-in definition for **post_randomize()**:

```
function void post_randomize;
   if (super) super.post_randomize();
   // Optional programming after randomization goes here.
endfunction
```

When *obj*.**randomize()** is invoked, it first invokes **pre_randomize()** on *obj* and also all of its random object members that are enabled. **pre_randomize()** then recursively calls **super.pre_randomize()**. After the new random values are computed and assigned, **randomize()** invokes **post_randomize()** on *obj* and also all of its random object members that are enabled. **post_randomize()** then recursively calls **super.post_randomize().**

Users may overload the **pre_randomize()** in any class to perform initialization and set pre-conditions before the object is randomized.

Users may overload the **post_randomize()** in any class to perform cleanup, print diagnostics, and check post-conditions after the object is randomized.

If these methods are overloaded, they must call their associated parent class methods, otherwise their pre- and post-randomization processing steps will be skipped.

Notes:
— Random variables declared as static are shared by all instances of the class in which they are declared. Each time the **randomize()** method is called, the variable is changed in every class instance.
— If **randomize()** fails, the constraints are infeasible and the random variables retain their previous values.
— If **randomize()** fails **post_randomize()** is not be called.
— The **randomize()** method may not be overloaded.
— The **randomize()** method implements object random stability. An object can be seeded by the **$srandom()** system call, specifying the object in the second argument.

14 Inline Constraints - randomize() with

By using the **randomize() with** construct, users can declare inline constraints at the point where the **randomize()** method is called. These additional constraints are applied along with the object constraints.

The syntax for **randomize() with** is:

result = *object_name*.randomize() **with** *constraint_block*;

object_name is the name of an instantiated object.

The anonymous *constraint block* contains additional inline constraints to be applied along with the object constraints declared in the class.

For example:

```
class SimpleSum
   rand bit[7:0] x, y, z;
   constraint c {z == x + y;}
endclass

task InlineConstraintDemo(SimpleSum p);
   int success;
   success = p.randomize() with {x < y;};
endtask
```

This is the same example used before, however, **randomize() with** is used to introduce an additional constraint that x < y.

The **randomize() with** construct can be used anywhere an expression can appear. The constraint block following **with** can define all of the same constraint types and forms as would otherwise be declared in a class.

The **randomize() with** constraint block may also reference local variables and task and function parameters, eliminating the need for mirroring a local state as member variables in the object class. The scope for variable names in a constraint block, from inner to outer, is: **randomize() with** object class, automatic and local variables, task and function parameters, class variables, variables in the enclosing scope. The **randomize() with** class is brought into scope at the innermost nesting level.

For example, see below, where the **randomize() with** class is “Foo.”

```
class Foo;
   rand integer x;
endclass

class Bar;
   integer x;
   integer y;

   task doit(Foo f, integer x, integer z);
      int result;
      result = f.randomize() with {x < y + z;};
   endtask
endclass
```

In the “f.randomize() with” constraint block, x is a member of class Foo, and hides the x in class Bar. It also hides the x parameter in the doit() task. y is a member of Bar. z is a local parameter.

15 Disabling Random Variables

The **$rand_mode()** system task can be used to control whether a random variable is active or inactive. When a random variable is inactive, it is treated the same as if it had not been declared **rand** or **randc.** Inactive variables are not randomized by the **randomize()** method, and their values are treated as state variables by the solver. All random variables are initially active.

15.1 $rand_mode()

The syntax for the **$rand_mode()** subroutine is:

task $rand_mode(**ON | OFF,** *object* [*.random_variable*]);

or

function int $rand_mode(*object***.***random_variable*);

object is any expression that yields the object handle in which the random variable is defined.

random_variable is the name of the random variable to which the operation is applied. If it is not specified, the action is applied to all random variables within the specified object.

The first argument to the **$rand_mode** task determines the operation to be performed:

Constant	Value	Description
OFF	0	Sets the specified variables to inactive so that they are not randomized on subsequent calls to the **randomize()** method.
ON	1	Sets the specified variables to active so that they are randomized on subsequent calls to the **randomize()** method.

For array variables, *random_variable* can specify individual elements using the corresponding index. Omitting the index results in all the elements of the array being affected by the call.

If the variable is an object, only the mode of the variable is changed, not the mode of random variables within that object (see Global Constraints in Section 11).

A compiler error is issued if the specified variable does not exist within the class hierarchy or it exists but is not declared as **rand** or **randc.**

The function form of **$rand_mode()** returns the current active state of the specified random variable. It returns 1 if the variable is active (ON), and 0 if the variable is inactive (OFF).
The function form of **$rand_mode()** only accepts scalar variables, thus, if the specified variable is an array, a single element must be selected via its index.

Example:

```
class Packet;
   rand integer source_value, dest_value;
   ... other declarations
endclass

int ret;
Packet packet_a = new;
   // Turn off all variables in object.
$rand_mode(OFF, packet_a);

   // ... other code
   // Enable source_value.
$rand_mode(ON, packet_a.source_value );

ret = $rand_mode( packet_a.dest_value );
```

This example first disables all random variables in the object *packet_a*, and then enables only the *source_value* variable. Finally, it sets the *ret* variable to the active status of variable *dest_value*.

16 Disabling Constraints

The **$constaint_mode()** system task can be used to control whether a constraint is active or inactive. When a constraint is inactive, it is not considered by the **randomize()** method. All constraints are initially active.

16.1 $constraint_mode()

The syntax for the **$constraint_mode()** subroutine is:

task $constraint_mode(**ON** | **OFF**, *object* [.*constraint_name*]);

or

function int $constraint_mode(*object***.** *constraint_name*);

object is any expression that yields the object handle in which the constraint is defined.

constraint_name is the name of the constraint block to which the operation is applied. The constraint name can be the name of any constraint block in the class hierarchy. If no constraint name is specified, the operation is applied to all constraints within the specified object.

The first argument to the **$constraint_mode** task determines the operation to be performed:

Constant	Value	Description
OFF	0	Sets the specified constraint block to active so that it is considered by subsequent calls to the **randomize()** method.
ON	1	Sets the specified constraint block to inactive so that it is not enforced on subsequent calls to the **randomize()** method.

A compiler error is issued if the specified constraint block does not exist within the class hierarchy**.**

The function form of **$constraint_mode()** returns the current active state of the specified constraint block. It returns 1 if the constraint is active (ON), and 0 if the constraint is inactive (OFF).

Example:
```
class Packet;
    rand integer source_value;
    constraint filter1 { source_value > 2 * m; }
endclass

function integer toggle_rand( Packet p );
    if( $constraint_mode( p.filter1 ) )
        $constraint_mode( OFF, p.filter1 );
    else
        $constraint_mode( ON, p.filter1 );

    toggle_rand = p.randomize();
endfunction
```

In this example, the toggle_rand function first checks the current active state of the constraint filter1 in the specified Packet object *p*. If the constraint is active then the function will deactivate it; if it's inactive the function will activate it. Finally, the function calls the randomize method to generate a new random value for variable *source_value*.

17 Static Constraint Blocks

Constraint blocks can be defined as static by including the **static** keyword in their definition.

The syntax to declare a static constraint block is:

static constraint *constraint_name* **{** *contraint_expressions* **}**

If a constraint block is declared as **static**, then calls to **$constraint_mode()** affect all instances of the specified constraint in all objects. Thus, if a static constraint is set to OFF, it is OFF for all instances of that particular class.

18 Dynamic Constraint Modification

There are several ways to dynamically modify randomization constraints:

— Implication and if-else style constraints allow declaration of predicated constraints.

— Constraint blocks can be made active or inactive using the **$constraint_mode()** system task. Initially, all constraint blocks are active. Inactive constraints are ignored by the **randomize()** function.

— Random variables can be made active or inactive using the **$rand_mode()** system task. Initially, all **rand** and **randc** variables are active. Inactive variables are ignored by the **randomize()** function.

— The weights in a **dist** constraint can be changed, affecting the probability that particular values in the set are chosen.

19 Random Number System Functions

19.1 $urandom

The system function **$urandom** provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The number is unsigned.

The syntax for **$urandom** is:

function unsigned int $urandom [(int *seed*)] ;

The *seed* is an optional argument that determines which random number is generated. The seed can be any integral expression. The random number generator generates the same number every time the same seed is used.

The random number generator is deterministic. Each time the program executes, it cycles through the same random sequence. This sequence can be made non-deterministic by seeding the **$urandom** function with an extrinsic random variable, such as the time of day.

For example:
```
    bit [64:1] addr;

    $urandom( 254 );                  // Initialize the generator
    addr = {$urandom, $urandom };     // 64-bit random number
    number = $urandom & 15;           // 4-bit random number
```

The **$urandom** function is similar to the **$random** system function, with two exceptions. **$urandom** returns unsigned numbers and it's automatically thread stable (see Section 20.2).

19.2 $urandom_range()

The **$urandom_range()** function returns an unsigned integer within a specified range.

The syntax for **$urandom_range** is:

function unsigned int $urandom_range(**unsigned int** *maxval*, **unsigned int** *minval* = 0);

The function returns an unsigned integer in the range *maxval* .. *minval*.

Example: `val = $urandom_range(7,0);`

If *minval* is omitted, the function returns a value in the range *maxval* .. 0.

Example: `val = $urandom_range(7);`

If *maxval* is less than *minval*, the arguments are automatically reversed so that the first argument is larger than the second argument.

Example: `val = $urandom_range(0,7);`

All of three previous examples produce a value in the range of 0 to 7, inclusive.

$urandom_range() is automatically thread stable (see Section 20.2).

19.3 $srandom()

The system function **$srandom()** allows manually seeding the RNG of objects or threads.

The syntax for the **$srandom()** system task is:

task **$srandom**(**int** *seed*, [object *obj*]);

The **$srandom()** system task initializes the local random number generator using the value of the given seed. The optional object argument is used to seed an object instead of the current process (thread). The top level randomizer of each **program** is initialized with **$srandom**(1) prior to any randomization calls.

20 Random Stability

The Random Number Generator (RNG) is localized to threads and objects. Because the stream of random values returned by a thread or object is independent of the RNG in other threads or objects, this property is called *Random Stability*. Random stability applies to:

— the system randomization calls, **$urandom**, **$urandom_range(),** and **$srandom()**.
— the object randomization method, **randomize()**.

Test-benches with this feature exhibit more stable RNG behavior in the face of small changes to the user code. Additionally, it enables more precise control over the generation of random values by manually seeding threads and objects.

20.1 Random Stability Properties

Random stability encompasses the following properties:

— Thread stability

Each thread has an independent RNG for all randomization system calls invoked from that thread. When a new thread is created, its RNG is seeded with the next random value from its parent thread. This property is called "hierarchical seeding."

Program and thread stability is guaranteed as long as thread creation and random number generation is done in the same order as before. When adding new threads to an existing test, they can be added at the end of a code block in order to maintain random number stability of previously created work.

— Object stability

Each class instance (object) has an independent RNG for all randomization methods in the class. When an object is created using **new**, its RNG is seeded with the next random value from the thread that creates the object.

Object stability is guaranteed as long as object and thread creation, as well as random number generation is done in the same order as before. In order to maintain random number stability, new objects, threads and random numbers can be created after existing objects are created.

— Manual seeding

All RNG's can be manually seeded. Combined with hierarchical seeding, this facility allows users to define the operation of a subsystem (hierarchy sub-tree) completely with a single seed at the root thread of the system.

20.2 Thread Stability

Random values returned from the **$urandom** system call are independent of thread execution order. For example:

```
integer x, y, z;
fork             //set a seed at the start of a thread
   begin  $srandom(100); x = $urandom;  end
                 //set a seed during a thread
   begin  y = $urandom; $srandom(200);  end
                 // draw 2 values from the thread RNG
   begin  z = $urandom + $urandom ;     end
join
```

The above program fragment illustrates several properties:

— Thread Locality. The values returned for x, y and z are independent of the order of thread execution. This is an important property because it allows development of subsystems that are independent, controllable, and predictable.

— Hierarchical seeding. When a thread is created, its random state is initialized using the next random value from the parent thread as a seed. The three forked threads are all seeded from the parent thread.

  Each thread is seeded with a unique value, determined solely by its parent. The root of a thread execution subtree determines the random seeding of its children. This allows entire subtrees to be moved, and preserve their behavior by manually seeding their root thread.

20.3 Object Stability

The **randomize()** method built into every class exhibits *object stability*. This is the property that calls to **randomize()** in one instance are independent of calls to **randomize()** in other instances, and independent of calls to other randomize functions.

For example:

```
class Foo;
   rand integer x;
endclass

class Bar;
   rand integer y;
endclass
```

```
    initial begin
        Foo foo = new();
        Bar bar = new();
        integer z;
        void = foo.randomize();
        // z = $random;
        void = bar.randomize();
    begin
```

— The values returned for foo.x and bar.y are independent of each other.
— The calls to randomize() are independent of the $random system call. If one uncomments the line "z = $random" above, there is no change in the values assigned to foo.x and bar.y.
— Each instance has a unique source of random values that can be seeded independently. That random seed is taken from the parent thread when the instance is created.
— Objects can be seeded at any time using the **$srandom**() system task with an optional object argument.

```
        class Foo;
            function void new (integer seed);
                    //set a new seed for this instance
              $srandom(seed, this);
           endfunction
        endclass
```

Once an object is created there is no guarantee that the creating thread can change the object's random state before another thread accesses the object. Therefore, it is best that objects self-seed within their **new** method rather than externally.

An object's seed may be set from any thread. However, a thread's seed can only be set from within the thread itself.

21 Manually Seeding Randomize

Each object maintains its own internal random number generator, which is used exclusively by its **randomize()** method. This allows objects to be randomized independent of each other and of calls to other system randomization functions. When an object is created, its random number generator (RNG) is seeded using the next value from the RNG of the thread that creates the object. This process is called hierarchical object seeding.

Sometimes it is desirable to manually seed an object's RNG using the **$srandom()** system call. This can be done either in a class method, or external to the class definition:

internally:

```
    class Packet;
        rand bit[15:0] header;
        ...
        function void new (int seed);
```

```
          $srandom(seed, this);
      ...
      endtask
   endclass
```

or externally:

```
   Packet p = new(200);       // Create p with seed 200.
   $srandom(300, p);          // Re-seed p with seed 300.
```

Calling **$srandom()** in an object's **new()** function, assures the object's RNG is set with the new seed before any class member values randomized.

Appendix A Operator Preced e nce and Associativity

Table 2 below shows the precedence and associativity of all SystemVerilog operators, including the additional operators used by random constraints. The new operators are shown in Table 1.

Operator	Description
dist	Distribution
inside	Set Membership
=>	Implication

Table 1 Additional operators proposed for constraints

Operator	Associativity
() [] .	left
Unary ! ~ ++ -- + - & ~& | ~| ^ ~^	right
**	left
* / %	left
+ -	left
<< >> <<< >>>	left
< <= > >= **inside** **dist**	left
== != === !==	left
&	left
^ ~^	left
|	left
&&	left
||	left
=>	right
? :	right
= += -= *= /= %= &= |= ^= <<= >>= <<<= >>>=	none

Table 2 Operator precedence and associativity in SystemVerilog including new operators

Appendix B Keywords

Below is the list of all new keywords added by this proposal.†

before†	**inside**	**randc**	**with**
constraint	**rand**	**solve**†	

† The keywords **solve** and **before** need not be implemented as keywords, but as context recognized identifiers.