Draft Standard for Verilog Randomization and Constraints Extensions

Copyright © 2003 by Cadence Design Systems, Inc.

This document is an unapproved draft of a proposed IEEE Standard. As such, this document is subject to change. USE AT YOUR OWN RISK! Because this is an unapproved draft, this document must not be utilized for any conformance/compliance purposes.

Table of Contents

1. Introduction 4

1.1 Scope 4

1.2 Purpose 4

1.3 Requirements 4

1.3.1 Support randomization for all Verilog 1364-2001 datatypes in an infrastructure that is extensible to future data type enhancements. 4
1.3.2 Express constraint expressions in Verilog syntax........ 4
1.3.3 Express creation of user-defined distribution 5
1.3.4 Stable/Reproducible stimulus 5
1.3.5 Control the randomization of variables and expressions involved in constraints 5
1.3.6 Obtain values in a specific order from the set of legal values expressed using constraints 5

2. Randomization System Tasks 5

2.1 Randomization Extensions to Define Constraints 5

2.1.1 $vr_constraint: Add a constraint on a set of variables, using an expression 5
2.1.2 $vr_set_weight (): Specify explicit weights for values of a constrained variable 6
2.1.3 $vr_keep_only: Specify a range of legal values 6
2.1.4 $vr_keep_out: Specify a range of values the variable cannot take 7
2.1.5 $vr_reset_distribution: Reset distributions for a constrained variable 7
2.1.6 $vr_set_mode: Specify a value generation mode for a constrained variable 7
2.1.7 $vr_set_seed: Specify explicit seed management for a constrained variable 8
2.1.8 $vr_set_global_seed: Specify the global seed 8

2.2 $vr_next: Generate next values of constrained objects 8

2.3 $vr_enable/disable_randomization: Enable/Disable randomization 9

3. Basic Randomization 9

3.1 Creating randomizable objects 9

3.1.1 Manipulating arbitrary data types 10

3.2 Generating random values 11

4. Stream and Seed Manipulation 12

4.1 Explicit seed manipulation 12

4.2 Global seed 12

4.3 Implicit seed manipulation 13

5. Constraint Specification and Constrained Randomization 13

5.1 Specifying constraint expressions 13

5.2 Operators supported in constraint expressions 14

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

5.3 Gene rating random values for multiple dependent objects ... 14

5.4 Semantics for multiple constraint expressions ... 14

5.5 Dynamic v/s static manipulation of constraints ... 14

5.6 Constraints on bit or part selects ... 15

6. Weight Specification and Biased Randomization ... 15

6.1 Specifying weights for particular values ... 15

6.2 Specifying weights for ranges of values ... 15

6.3 Specifying simple distributions ... 16

6.3.1 Specify legal set of values ... 16

6.3.2 Specify illegal range of values ... 16

6.4 Combining constraints and distributions ... 17

7. Controlling randomization state ... 18

7.1 Enable/disable randomization ... 18

7.2 Extracting constrained random values ... 18

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

Randomization and Constraints Extensions

1. Introduction

1.1 Scope

This proposal extends Verilog 1364-2001 to efficiently do constrained random stimulus generation. It provides ways to specify constraints and generate values to obtain both uniform and biased distributions for Verilog data type objects.

1.2 Purpose

Verilog allows modeling designs at different levels of abstraction from switch level up to behavioral RTL. For designs at all these levels it is desirable to build directed and constrained random testbenches to verify the design. With the increases in design complexity, the verification effort is becoming more time consuming. The ability to efficiently generate constrained random stimulus has become an integral part of the verification process.

The Verilog 1364-2001 standard provides a $random system task for generating fixed size unconstrained uniformly distributed random values. It also provides very basic seed control by passing a seed value as an additional argument to the $random system task.

The proposed extensions support modeling constraints and biasing values to generate both signal level and transaction level reproducible constrained random tests. These extensions use existing Verilog 1364-2001 concepts and introduce a set of system tasks and functions to provide this functionality.

1.3 Requirements

1.3.1 Support randomization for all Verilog 1364-2001 datatypes in an infrastructure that is extensible to future data type enhancements.

The Verilog type system allows two basic kinds of objects, variables and nets. The extensions should allow the generation of constrained random values for variables, and the specification of constraints in terms of nets and variables.

The solution should also be extensible. As new data types are added to the language the solution should support constrained randomization on these types.

1.3.2 Express constraint expressions in Verilog syntax

Constraints for randomization should allow the expression of any combination of legal values for single or multiple dependent variables.

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

The proposed solution should allow the modeling of an expression representing a non-temporal constraint for the random value generation process.

The expression should be expressed in Verilog syntax and syntactically analyzed by the Verilog language parser.

1.3.3 Express creation of user-defined distribution

The proposal should allow specifying bias for legal values in order to generate a certain set of legal values more often than others. This will allow the creation of any specific distribution to be generated.

1.3.4 Stable/Reproducible stimulus

The ability to generate reproducible stimulus across multiple runs of the test and to allow incremental changes to the test while retaining its reproducible behavior is critical for random tests in a verification test bench. Thus the proposal should allow a means to support incremental additions to the test and have reproducible tests across different simulation runs.

1.3.5 Control the randomization of variables and expressions involved in constraints

Under certain test scenarios it is desirable to provide controls for turning randomization of variables on or off based on test conditions. This is particularly important for dependent variables and variables that are part of composite data types.

1.3.6 Obtain values in a specific order from the set of legal values expressed using constraints

Constraints express legal sets of values for variables. The random value generation process should allow generating random values in certain specific orders. It is often desirable that a test generates legal values in a particular order, for example cycle through all possible values before generating the same legal value again.

2. Randomization System Tasks

2.1 Randomization Extensions to Define Constraints

The tasks that are described in this section can be invoked individually on variables from within sequential code, or they can be invoked in the initial block of a Verilog module.

2.1.1 $vr_constraint: Add a constraint on a set of variables, using an expression

$vr_constraint() system task can be used to add a constraint expression

$vr_constraint(<*Verilog_expression*> **)**

The argument to this task is a Verilog expression which is used by the constraint solver to generate constrained random values. The expression corresponds to a constraint that is satisfied if the expression

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

evaluates to a non-zero value. The variables in the expression become dependant variables, subject to randomization. Nets cannot be randomized.

Constraints on a variable should be specified before random values are generated for that variable. Therefore, all $vr_constraint calls on a variable should be made before the first $vr_next call on that variable.

2.1.2 $vr_set_weight (): Specify explicit weights for values of a constrained variable

$vr_set_weight() system task specifies weights for values of a constrained variable. It has the following two usage scenarios:

With three arguments, it specifies the weight for specific value of a variable:

```
$vr_set_weight( <variable>,        // The constrained variable, of type reg, integer, real
                <value>, // The value that is to be weighted, same type as <variable>
                <weight> )         // The weight, an integer >= 0
```

With four arguments, it specifies the weight for a specific range of values for a variable:

```
$vr_set_weight(  <variable>,              // The constrained variable
                 <lower_bound_value>,     // The lower bound of the range of values
                 <upper_bound_value>,     // The upper bound of the range
                 <weight> )               // The weight, an integer >= 0
```

If the sum of all weights on a variable does not add to 100, then the percentage is taken to be a ratio of the weight divided by the sum of all weights on a variable.

$vr_set_weight can be called any time during simulation.

2.1.3 $vr_keep_only: Specify a range of legal values

$vr_keep_only() modifies the current distribution for the supplied variable. Its effect is cumulative to other $vr_keep_only and $vr_keep_out calls. It has the following two usage scenarios:

With two arguments specifies an individual legal value specified as second argument to the system task.

```
$vr_keep_only(  <variable>,              // The constrained variable, of type reg, integer, real
                <value>)                 // legal value to be included, same type as <variable>
```

With three arguments specifies the range of values including the lower_bound and the upper_bound values to be included.

```
$vr_keep_only(  <variable>,                    // The constrained variable
                <lower_bound>,                 // The lower bound of the range of values
                <upper_bound> )                // The upper bound of the range of values
```

If multiple $vr_keep_only ranges are specified for a given variable, then the result is the intersection of the specified ranges.

$vr_keep_only can be called any time during simulation.

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

2.1.4 $vr_keep_out: Specify a range of values the variable cannot take

$vr_keep_out() modifies the current distribution for the supplied variable. It's effect is cumulative to other $vr_keep_only and $vr_keep_out calls.

With two arguments specifies an individual value to be excluded.

```
$vr_keep_out(   <variable>,        // The constrained variable, of type reg, integer
                 <value > )        //  value to be excluded, same type as <variable>
```

With three arguments specifies the range of values including the lower_bound and the upper_bound values to be excluded.

```
$vr_keep_out(   <variable>,        // The constrained variable
                <lower_bound>,     // The lower bound of the  range of values
                 <upper_bound> )   // The upper bound of the range of values
```

If multiple $vr_keep_out values and ranges are specified for a given variable, then the result is the accumulation of all the values and ranges. If $vr_keep_only calls have also been made for the variable, then the $vr_keep_out values are removed from the set of values specified in the $vr_keep_only calls. So, to determine the resulting set of values after multiple $vr_keep_out and $vr_keep_only calls, take the intersection of all $vr_keep_only ranges, and remove from this the $vr_keep_out values and ranges.

$vr_keep_out can be called any time during a simulation.

2.1.5 $vr_reset_distribution: Reset distributions for a constrained variable

The $vr_reset_distribution() system task sets the attached distributions (from $vr_keep_only()s, $vr_keep_out()s, or $vr_set_weight()) to empty, and sets the value generation mode to "RANDOM".

```
$vr_reset_distribution( <variables> ) // specify one or more variables as arguments
                                      // can be type reg, integer, real, or array of these
```

$vr_reset_distribution can be called any time during the simulation.

2.1.6 $vr_set_mode: Specify a value generation mode for a constrained variable

$vr_set_mode() sets the value generation mode for a constrained variable.

```
$vr_set_mode(    <variable>,                  // The constrained variable, reg, integer, real or array
                <value_generation_mode>)      // The value generation mode
```

Where <value_generation_mode> can be one of the following strings:

- "RANDOM": uses a uniform distribution across the range of all legal values (low overhead)
- "SCAN": keeps a history and starts with the smallest legal values (medium overhead)
- "RANDOM_AVOID_DUPLICATE": keeps a history and avoids duplicated values until all legal values have been generated, resets history (high overhead)

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

- "DISTRIBUTION": takes a user-specified distribution, and uses it to bias the randomization process (overhead depends on the distribution). This mode is implied by calls to $vr_set_weight on a variable.

$vr_set_mode can be called any time during the simulation.

2.1.7 $vr_set_seed: Specify explicit seed management for a constrained variable

$vr_set_seed() system task specifies initial seed for constrained variable.

```
$vr_set_seed(   <variable>,        // The constrained variable, of type reg, integer, real or array
                <seed_value> )     // An integer representing seed value
```

$vr_set_seed should be called on a variable before constraints are specified on that variable, and before random values are generated for that variable.

2.1.8 $vr_set_global_seed: Specify the global seed

$vr_set_global_seed() system task specifies the global seed value for the entire simulation run.

$vr_set_global_seed(<global_seed_value> **)**

The global seed value is used to do implicit seed management.

The argument to the system task can be an explicit global seed, which is a 32-bit value, or it can be the string constant "PICK_RANDOM_SEED". The later allows the system task to pick a random seed value based on the current system time. The default value for the global seed is 1.

$vr_set_global_seed should be called before constraints are specified, and before random values are generated.

2.2 $vr_next: Generate next values of constrained objects

$vr_next() system task generates values for the constrained variables specified as arguments to the system task call. It has the following 2 scenarios:

Generate the next value for a set of constrained variables

```
$vr_next( <variables> )          // specify one or more variables as arguments to generate values.
                                 // can be type reg, integer, real, or array of these
```

Generate next values for all the variables in a module instance

```
$vr_next( <Verilog_module_instance> )     // generate next random values for all variables in the module
                                          // instance having explicit constraints
```

All explicit seed settings and constraint expressions should be specified before the first call to $vr_next for a given variable.

2.3 $vr_enable/disable_randomization: Enable/Disable randomization

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

$vr_disable_randomization**() and **$vr_enable_randomization() system tasks are used to disable and enable randomization respectively.

```
$vr_disable_randomization( <variables> )     // disable randomization on variables specified as
                                              // arguments
$vr_enable_randomization( <variables> )      // enable randomization on variables specified
                                              // as arguments, default in on
```

The variables in the $vr_disable_randomization call will not be updated when $vr_next is called. If this variable is used in a constraint expression, then the constraint solver will use the current value and will not change it. It is an error if the value assigned to the disabled variable is out of range.

Randomization for a given variable can be enabled or disabled at any time during simulation.

3. Basic Randomization

Generating random values requires specifying the type of the random data. This proposal uses any Verilog data object as a randomizable object. It can be determined at the runtime which objects are involved in randomization calls.

Data objects of arbitrary Verilog data types can be randomized through the use of the proposed randomization extensions. For example an integer variable declared in a Verilog module can be randomized by using the following code:

```
integer data;
$vr_next(data);
```

3.1 Creating randomizable objects

Any Verilog variable object can be randomized using the $vr_next() system task. Every Verilog object involved in randomization calls will have an internal random stream attached with it to perform randomization.

For example, the following code generates random values for the num_samples, address and data objects respectively:

```
reg[3:0] num_samples;
reg [63:0] address;
reg[63:0] data;
$vr_next(num_samples);
for ( i = 0 ; i < num_samples; i = i + 1) begin
   $vr_next(address);
  $vr_next(data);
end
```

Similarly random values can be generated for arrays and for module instances.

```
integer payload[15:0];
$vr_next(payload);          //  generate new random values for all the 16 elements on the
                            //  payload integer array type data object.

module packet;
```

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

```
 reg[7:0] source;
reg[7:0] destination;

initial begin
   $vr_keep_only(source, 1, 96);
   $vr_keep_out(destination, 16, 64);
end
endmodule;

packet p1();
$vr_next(p1);                    // generate new random values for all the fields of p1 instance
```

3.1.1 Manipulating arbitrary data types

Verilog supports different data kinds and types to model hardware. Nets represent connections between hardware elements. Registers represent data storage elements. Just as in real circuits, nets have values continuously driven on them by the outputs of connected. Registers retain value until another value is placed onto them. An integer is a general-purpose register data type used for manipulating signed quantities.

This section explains the semantics of randomizing these different data types.

3.1.1.1 Register variables

Registers of arbitrary data width can be subject to constrained randomization. As specified by the language these will be treated as unsigned quantities and manipulated as such in the randomization context. For example the following code results in random values for the 128 bit register argument:

```
reg [127:0] data;
$vr_next(data);
```

3.1.1.2 Integer data type

Integers can also be subject to constrained randomization. As specified by the language these will be treated as signed quantities and manipulated as such in the randomization context. For example the following code results in signed integer random values for integer variable value:

```
integer value;
$vr_next(value); // will result in signed values between [INT_MIN, INT_MAX]
```

where INT_MIN and INT_MAX are defined by the size of the integer variable depending on the underlying size for integer type used by the simulator.

3.1.1.3 Nets

Nets have values continuously driven on them by the outputs of connected devices. Nets cannot be randomized.

On the other hand objects of a net type can be used to control value generation of other variable type objects. For that purpose, these objects can be involved in constraints to influence the value generation of

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

dependent variables but not be allowed to be randomized themselves. For example the following code will result in an error while trying to generate value for the net:

```
reg [7:0] data_reg;
wire [7:0] data_net;

$vr_constraint(data_reg > data_net); // valid constraint (involving data_reg and data_net)

$vr_next(data_reg); // OK to generate value for reg
$vr_next(data_net); // error for trying to generate random value for net
```

The call to $vr_next(data_reg) will generate a new random value for data_reg based on the current value for data_net.

3.2 Generating random values

The $vr_next() system task takes a variable number of Verilog data type objects as arguments. Objects of valid randomizable types result in new random values to be generated for the objects.

Based on the type of constraints and the number of arguments specified to $vr_next(), the constraints will be solved simultaneously or independently while using the dependent values for other variables.

If a module instance is passed as an argument to $vr_next() it will generate new random values for all randomizable objects recursively in that module instance. Note that only variables having explicit constraints will be randomized in this manner. If no explicit constraint is specified on a variable , it will not be randomized unless it is the argument to a $vr_next call.

For example the following code shows the various scenarios of using $vr_next() to generate new random values:

Solving constraints for dependent variables with one call to $vr_next() or multiple calls to $vr_next().

```
reg[7:0] addr;
reg[7:0] data;

addr = 67; data = 0;
$vr_constraint(addr >= 64 && addr <= 127);
$vr_constraint(addr > data);

$vr_next(addr); // generates new random value for addr considering data == 0
$vr_next(data); // generates new random value for data considering value generated for
                // addr in the previous statement
$vr_next(addr, data); // generates new random values for both addr and data
                      // satisfying the constraints
```

Following code shows example of doing randomization on variables in a module instance, with explicit constraints.

```
module packet_constraint;
reg [7:0] src;
reg [7:0] dest;
reg [2:0] ptype;
```

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

```
initial
begin
   $vr_keep_only(src, 16, 96);
   $vr_keep_only(dest, 0, 226);
end
endmodule

packet_constraint pc(); // instance of module packet_constraint
$vr_next(pc); // generates values for all variables having explicit constraints
              // ptype in the above example in $vr_next(pc) will not generate new random value
              // as it is not having any constraints specified.
$vr_next(pc.ptype); // explicit call to generate new value for pc.ptype will result in generating a new
                    // unconstrained random value for pc.ptype
```

4. Stream and Seed Manipulation

In order to support advanced seed management, enable reproducibility and manage randomization for multiple different independent objects we introduce the notion of a random stream. Every independent random object will have a random stream attached to it to perform randomization. Dependent objects will share the same random stream unless otherwise specified.

This stream provides a series of random unsigned integer values for randomization of the object to which it is attached. It can take an explicit seed from the user, or extract a seed from the design context. It uses the same algorithm as **jrand48()** from the standard C library.

4.1 Explicit seed manipulation

An initial seed for the random streams attached to the randomizable object can be set by calling the system task $vr_set_seed(). This call should be made before any call to the $vr_next() system task.

For example the following code will set an initial seed value for the random stream attached to the variable data:

```
integer data;

// specify initial seed value to 200 for random stream attached with data
// note the initial value should be set before any calls to $vr_next() or setting
// other constraints such as weights.
$vr_set_seed(data, 200);                    .
for (i = 0; i < 10; i = i + 1) begin
   $vr_next(data);
end
```

4.2 Global seed

The randomization facility has one global seed that the user can manipulate using the $vr_set_global_seed() system task. If the user does not provide the global seed explicitly, the default global seed is 1.
For example the following code will set the global seed value :

```
$vr_set_global_seed(1000); // sets the global seed to 1000
                           // global seed influences the seed values for other random stream objects
```

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

```
$vr_set_global_seed("PICK_RANDOM_SEED"); // sets global seed to a seed value generated
                                         // using current system time.
```

4.3 Implicit seed manipulation

Each randomizable object is uniquely identified by its hierarchical name in the design. This proposal uses this unique hierarchical name of the object along with the global seed to generate a unique seed for each random object.

For example the following code will result in generating seed values for the variables involved in randomization, based on their hierarchical path name and also on the global seed value:

```
module data_gen;
integer data;
  $vr_next(data); // will result in seed value generated for data based on full hierarchical
                  // path name for this object. This value can be changed by adjusting the
                  // the global seed value
endmodule
```

5. Constraint Specification and Constrained Randomization

5.1 Specifying constraint expressions

Constraints are specified by passing a Verilog expression as an argument to the $vr_constraint() system task. The constraint expression has the same syntax and semantics as Verilog expressions. Multiple constraint expressions can be specified using multiple calls to the $vr_constraint() system task. The expressions involving dependent variables will have conjunction semantics; that is, the expressions given in multiple calls to $vr_constraint will be logically ANDed together.

For example the following code sets up constraints for variables addr and data:

```
reg [15:0] addr;
reg [15:0] data;

$vr_constraint (addr >= 127 && addr <= 4096); // specify legal address range values
$vr_constraint(data >= 100 && data <= 5000);  // specify legal data range values
```

The above two constraint expressions specify constraints on individual elements and do not specify any dependency. These will be treated as two separate expressions. This will not result in conjunction of the two expressions in eventual dependency checks. If either of the two expressions overconstrains the values such that no solution is possible, then that expression will be ignored and an error message generated. However, the expression which does not overconstrain the values will not be ignored.

Now consider the addition of the following constraint expression to the code:

```
$vr_constraint(addr > data);  // specify dependency of addr on data values
```

In this case, all the three constraint expressions will result in an expression which is the conjunction of all the above three expressions. Failure to satisfy either one or a combination of the expressions will result in overconstraint errors.

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

Note that all the constraints specified by $vr_constraint(expr) calls are hard constraints and must be satisfied under all conditions.

5.2 Operators supported in constraint expressions

All Verilog operators can be used in constraint expressions. Verilog precedence and associativity rules will be used for the expressions. The following operators will suffice to represent most constraints for randomization:

Arithmetic operators +, -, *
Relational operators ==, !=, >, >=, <, <=
Logical operators !, &&, ||
Conditional operator ? :

5.3 Generating random values for multiple dependent objects

The solver will simultaneously solve constraints for dependent variables if these are passed in a single call to $vr_next(). If $vr_next is called on a subset of the dependant variables, then the remaining variables will retain their existing value.

For example the following code,

```
initial begin
   $vr_constraint (a == b);
end

$vr_next(a);
$vr_next(b);
```

will always result in the same value for both ‘a’ and ‘b’. But if you generate the values for both ‘a’ and ‘b’ together in the same call then these will be solved simultaneously and that will result in different values satisfying the constraint “a == b”.

Replacing the above two calls to $vr_next() with one single call as following

```
$vr_next(a, b);
```

will generate new values for a and b which are the same but not always locked at the initial value.

5.4 Semantics for multiple constraint expressions

Different constraint expressions can be added using different calls to the $vr_constraint system task. The expressions involving dependent variables will be conjoined to result in a more complex constraint expression to be solved for all the variables involved in the expressions.

5.5 Dynamic v/s static manipulation of constraints

All the constraint expressions should be specified before the first call to the $vr_next() system task. Changes to constraints after the first call to $vr_next() will be ignored and a warning message generated.

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

5.6 Constraints on bit or part selects

Constraints can be specified on bit and part selects of the variables involved in the constraint specification. Constrained random values can only be generated for the entire object and not for bit or part selects of the object.

6. Weight Specification and Biased Randomization

While a constraint specifies the range of legal values, a weight specification biases the random value generation process so that some values are generated more often than others.

Distributions are specified using collection of weighted values or a collection of weighted ranges.

6.1 Specifying weights for particular values

The $vr_set_weight() system task is used to specify the weight for a particular value. E.g. to specify a distribution in which the value “1” is generated 60% of the time and the value “2” is generated 40% of the time, the following code can be used:

```
integer data;
$vr_set_weight(data, 1, 60);
$vr_set_weight(data, 2, 40);
for ( i = 0; i < 10; i = i + 1) begin
   $vr_next(data);
end
```

If the sum of all weights on a variable does not add to 100, then the percentage is taken to be a ratio of the weight divided by the sum of all weights on a variable.

6.2 Specifying weights for ranges of values

The $vr_set_weight() system task is used to specify the weights on a range of values too. For example, generating the range [0,1] 40% of the time and [2,10] 60% of the time can be achieved using the following code:

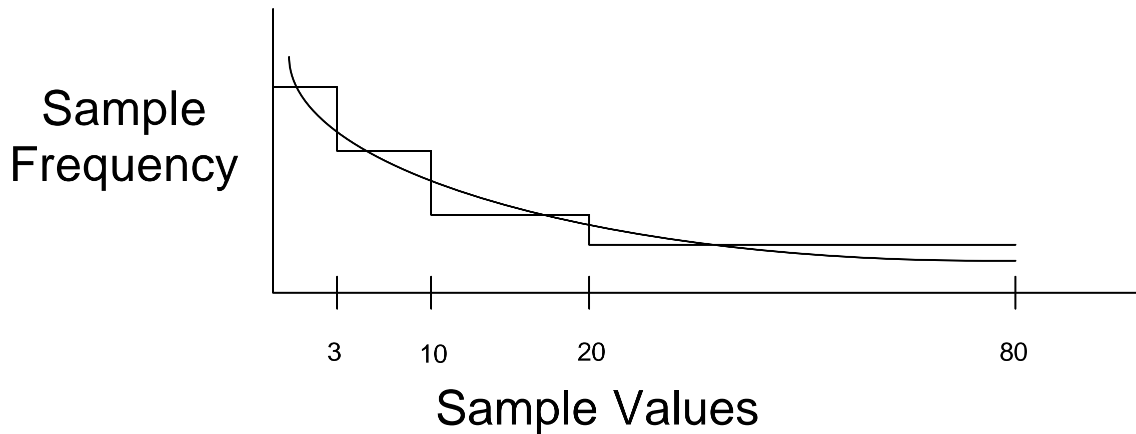
```
integer data;
$vr_set_weight(data, 0, 1, 40);
$vr_set_weight(data, 2, 10, 60);

for (i = 0; i < 10; i = i + 1) begin
   $vr_next(data);
end
```

In this example, the $vr_next() system task will select a range according to the weights, and then select a value from the range using a uniform probability distribution. As a result while the chance of having some value within the range [2,10] is higher than that for some value within the range [0,1], the chance of having the value 10 is much smaller than for the value 0. Effectively, the values 40 and 60 reflect the area under the range.

Using weights on ranges, a generic distribution, such as an exponential distribution, can be approximated by a step-like distribution, as shown in the following diagram:

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.



Using this step-like distribution, weights can be set on range of values and used to bias the randomization process:

```
integer data;

$vr_set_weight(data, 1,3, 100);
$vr_set_weight(data, 4,10, 30);
$vr_set_weight(data, 11,20, 20);
$vr_set_weight(data, 21,80, 80);
...

for ( i=0; i<20; I = I + 1)  begin
    $vr_next(data);
    process(data);
end
```

6.3 Specifying simple distributions

This proposal also present ways to specify simple constraints in the form of disjoint ranges of values.

The $vr_keep_only and $vr_keep_out system tasks provides a convenient way to specify distribution. The calls to $vr_keep_only and $vr_keep_out are cumulative, and are combined using a simple conjunction semantics.

6.3.1 Specify legal set of values

The $vr_keep_only() system task allows the user to modify the current distribution for a variable to include only the supplied value or range of values. Multiple $vr_keep_only calls can be used to specify multiple values or multiple ranges.

6.3.2 Specify illegal range of values

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

The $vr_keep_out() system task modifies the current distribution for a variable to exclude the supplied value or range of values , depending on the number of arguments to the system task.

For example the following code will result in values between [0,4] excluding 2:

```
integer data;

$vr_keep_only(data, -10, 10);        // specify [-10,10] to be legal values for data
$vr_keep_only(data, 0,4);            // specify [0,4] to be legal values for data
$vr_keep_out(data, 2);               // specify 2 to be excluded from the legal set
$vr_next(data); // generate a value among { 0, 1, 3, 4 }
```

6.4 Combining constraints and distributions

When $vr_next() is passed a data object with only distribution specified, a random value is generated with respect to the supplied distribution.

Consider the following examples involving distributions and constraints:

```
integer  a;

$vr_constraint( a < 1 );

$vr_set_weight(a, 10, 50);
$vr_set_weight(a, 11, 50);

$vr_next(a); // generate a value for “a” among { 10, 11 } with an error report about “a < 1” is made.
```

In this example, the constraint "a < 1" is checked after value generation from the distribution, and an error report is generated. The final value of *a* retains the value generated from the distribution.

When $vr_next() is executed on a module instance or multiple variables, the values are generated in two steps. The first step is to pick values for all variables with distribution mode turned on. The values are picked from the distribution directly without consulting the related constraints. If there are other fields without the distribution mode turned on, a second step is taken to analyze the constraint and generate a constrained random value for them, using the values generated in the previous step, i.e., as if randomization has been turned off via $vr_disable_randomization() for those variables in step 1.

For example:

```
integer a;
integer b;

$vr_constraint( a < b && a < 5 );

$vr_next(a,b); // generate values for a and b according to the constraint expression "a < b && a < 5".

$vr_set_weight(a, 2, 50);
$vr_set_weight(a, 4, 50);

$vr_next(a, b);
// step 1: generate a value for a among { 2, 4 } and the constraint "a < b && a < 5" is ignored.
```

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

```
// step 2: use the value generated in step 1 for “a”, and invoke the constraint solver to solve the
// constraint "a < b && a < 5", and create a random value for “b”,  while
// keeping the same value for “a”.

$vr_reset_distribution(a);
$vr_set_weight(a, 2, 50);
$vr_set_weight(a, 11, 50);

$vr_next(a, b);

// step 1: generate a value for a among { 2, 11 } and the constraint "a < b && a < 5"
// is ignored. Let's assume 11 is selected.
// step 2: use the value generated in step 1 for a, and invoke the constraint solver to solve the
// constraint expression "a < b && a < 5". In this case, since the value selected for “a”
// in step 1 violates the constraint, no legal value for “b” can be found to satisfy
// the constraint expression and an error is reported. A unconstrained value is selected
// for “b” in this case.
```

7. Controlling randomization state

The randomization facility can be configured at runtime in several ways.

7.1 Enable/disable randomization

$vr_enable_randomization() and $vr_disable_randomization() can be used to turn on and off randomization for a randomizable data object. The default is on.

7.2 Extracting constrained random values

The $vr_set_mode() system task allows specifying mode of extracting random values from the set of legal values specified using constraints. When randomization is on, the default value generation mode is “RANDOM”. For example the following code will generate values in the order specified by $vr_set_mode():

```
integer data;

$vr_keep_only(data, 1, 10);
$vr_keep_out(data, 4, 7);

$vr_set_mode(data, “RANDOM_AVOID_DUPLICATE”);
for (i = 0; i < 12; i = i + 1) begin
   $vr_next(data);                    // 3 , 8, 1, 9, 2, 10, 10, 2, 8, 9, 3, 1
end
```

In the above case all the values are exhausted before the next set of cyclic random values is generated.

```
$vr_set_mode(data, “SCAN”);
for (i = 0; i < 12; i = i + 1) begin
   $vr_next(data);                    // 1, 2, 3, 8, 9, 10, 1, 2, 3, 8, 9, 10
end
```

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.