Randomization

Constraints

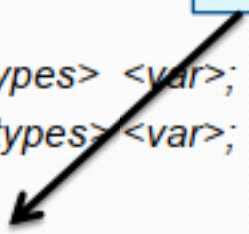
```
class  class_name;

<class_properties>;

rand   <sv_data_types>  <var>;
randc  <sv_data_types>  <var>;

constraint cons_name {
<constraints_expression_n_items> ;
}
<class_methods>;
endclass
```

The values of random variables are determined using constraint expressions that are declared using constraint blocks



Simple Class with Random Variables

```
class Transaction;
rand bit [7:0]   addr;
rand bit [31:0] data;

constraint valid {
addr inside {[0:15]};
data inside {[100:500]};
addr > 16;
}

endclass
```

```
program test;
Transaction pkt;
bit ret;
initial begin
pkt=new;
ret=pkt.randomize();
end
endprogram
```

```
pkt.addr=$urandom;
pkt.data=$urandom;
```

rand & randc variables

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

- Variables declared with the **randc** keyword are random-cyclic variables that cycle through all the values in a random permutation of their declared range.

randc variables

- **randc bit** [1:0] y;
- The variable y can take on the values 0, 1, 2, and 3 (range of 0 to 3).
- The basic idea is that **randc** randomly iterates over all the values in the range and that no value is repeated within an iteration.

```
class A;
randc bit [1:0] y;
endclass

program test;
A a;
initial begin
a=new;
repeat(12) begin
a.randomize();
$display("a.y=%0d",a.y);
end
end
endprogram
```

First permutation 0 -> 2 -> 1-> 3
Second permutation 3 -> 1 -> 0-> 2
Third permutation 2 -> 1 -> 0-> 3

Simple constraints

```
Class A;
rand bit [7:0] len ,addr ,src,sel;
rand bit [31:0] data;
rand bit wr,sel;

constraint my_cstr  {
addr  > 0;  addr < 15;
wr == 1;
data > 100 && data <500;
len <= src;
if (sel == 10) {
src inside {10, [30:40] , [66:88] ,100 };
} else {
src inside {[199:255]};
}
}
endclass
```

```
program test;
A a1;
bit ret;

initial begin
a1=new;
ret=a1.randomize();
end
endprogram
```

Set membership (inside operator)

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

Equivalent to a == b || a == c

```
int arr[4] = '{ 5, 10, 15, 20 };
rand int v1,v2;
constraint c3 { ! (v1 inside {arr};) }
constraint c4 { (v2 inside {arr};) }
```

v1!= 5 ,v1 !=10 ,v1 != 15 , v1 !=20

v2= 5 ,v2=10 ,v2= 15 , v2=20

```
rand bit [6:0] b;  // 0 to 127
rand bit [5:0] e;  // 0 to 63
constraint c_range {
b inside {[$:4], [20:$};
e inside {[$:4], [20:$};
}
endclass
```

0 <= b <= 4 || 20 <= b <= 127
0 <= e <= 4 || 20 <= e <= 63

Distributions using dist operator

```
class A;
rand bit [7:0] sa,da;

constraint c_dist {

sa dist {0 := 40, [1:3] := 60 } ;
// sa = 0, weight = 40/220
// sa = 1, weight = 60/220
// sa = 2, weight = 60/220
// sa = 3, weight = 60/220

da dist { 0 :/ 40, [1:3] :/ 60 };
// da = 0, weight = 40/100
// da = 1, weight = 20/100
// da = 2, weight = 20/100
// da = 3, weight = 20/100
}
endclass
```

*The **dist** operator allows you to create **weighted distributions** so that some values are chosen more often than others.*

```
program test;
A a1;
bit ret;

initial begin
a1=new;
repeat (220)
void'(a1.randomize();)
end
endprogram
```

Weighted Distributions

- *The dist operator allows you to create weighted distributions so that some values are chosen more often than others.*
- *The dist operator takes a list of values and weights, separated by the := or the :/ operator.*
- *The := operator specifies that the weight is the same for every specified value in the range.*
- *Whereas the :/ operator specifies that the weight is to be equally divided between all the values.*
- *A **dist** operation shall not be applied to **randc** variables*

Conditional Constraints

```
typedef enum {SHORT , MEDIUM , JUMBO} pkt_type_e;
class A;
 rand pkt_type_e pkt_size;
 rand int len;

constraint c_len_frames {
if (pkt_size == JUMBO)
len  inside {[2000:5000]};
else if (pkt_size == SHORT)
len  < 64 ;
else
len  inside {[64:1024]};  //MEDIUM
}
endclass
```

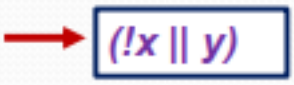
```
program test;
A a1;
bit ret;

initial begin
a1=new;
ret=a1.randomize();
end
endprogram
```

implication

```
class A;
rand bit x;           // 0 or 1
rand bit [1:0] y;  // 0, 1, 2, or 3

constraint c_xy {
(x==0) -> y==0;
}
endclass
```




```
program test;
A a1;
bit ret;

initial begin
a1=new;
ret=a1.randomize();
end
endprogram
```

- Implication operator says that **when x==0, y is forced to 0**.
- y can take any value 0,1,2,3 when x=1.

- However, implication is **bidirectional** in that if y were forced to a nonzero value, x would have to be 1.

solve...before

```
class A;
rand bit x;
rand bit [1:0] y;

constraint c_xy {
(x==0) -> y==0;
solve x before y;
}
endclass
```

The solve-before constraints provide a mechanism for ordering variables so that X can be chosen independently of Y.

In this case, the order constraint instructs the solver to solve for x before solving for y.

Adding this order constraint does not change the set of legal value combinations, but alters their probability of occurrence

The solver chooses values of x (0, 1) with equal probability.
In 1,000 calls to randomize(),
x is 0 about 500 times, and 1 about 500 times.

- When x is 0, y must be 0.
- When x is 1, y can be 0, 1, 2, or 3 with equal probability

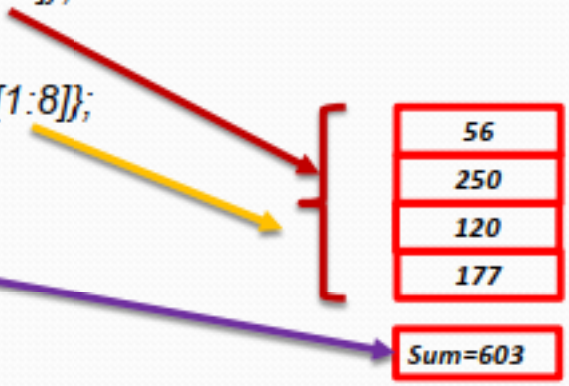
Constraining individual array elements

```
class A;
rand bit [31:0]  dyn_arr[ ]; //dynamic array
constraint c_len {

foreach (dyn_arr[i])
dyn_arr[i] inside {[1:255]}; //Each element must be in the range 1 to 255

dyn_arr.size() inside {[1:8]};

dyn_arr.sum < 1024;
}
endclass
```



constraint_mode

- A class can contain multiple constraint blocks.
- At run-time, you can use the built-in **constraint_mode()** routine to turn constraints on and off.

```
class Packet;
rand bit [15:0] length;

constraint  c_long   {
length inside {[1000:1900]};
}

constraint c_short   {
length inside {[1:32]};
}

endclass
```

```
Packet pkt;
initial begin
pkt=new;
pkt.c_long.constraint_mode(0);
void'(pkt.randomize());
#100
pkt.c_long.constraint_mode(1);
pkt.c_short.constraint_mode(0);
void'(pkt.randomize());
#100
pkt.constraint_mode(0);
void'(pkt.randomize());
end
```

Disabling random variables with rand_mode()

- The rand_mode() method can be used to control whether a random variable is active or inactive.

```
class A;
rand int x;
rand int y;
rand int z;
endclass
```

```
A a1;
initial begin
a1=new;
a1.rand_mode(0);
a1.x.rand_mode(1);
void'(a1.randomize());
#100 a1.rand_mode(1);
void'(a1.randomize());
end
```

Turn off rand mode on all variables of object a1

Turn on rand mode on variable x of object a1

Turn on rand mode on all variables of object a1

checker

```
class Packet;

  bit [7:0] sa,da;

 constraint valid {
  sa inside {[1:8]};
  da inside {[1:8]};
 }

endclass
```

```
program test;

Packet pkt;

initial begin
pkt=new;

pkt.sa=4;pkt.da=3;

void'(pkt.randomize());

end

endprogram
```

Error : Constraints are inconsistent and cannot be solved

In-line Constraints

```
class Transaction;
rand bit [31:0] addr, data;
constraint c1 {
addr inside { [0:100],[1000:2000] } ;
}
endclass
```

In-line constraints adds additional constraint to any existing constraints in class.

```
constraint c1 {
addr inside { [0:100],[1000:2000] } ;
}

constraint c_with {
addr >= 50; addr <= 1500;
data < 10;
}
```

```
Transaction t;
bit [31:0] data;
initial begin
t = new();
// addr is 50-100, 1000-1500, data < 10
void'( t.randomize() with {addr >= 50; addr <= 1500; data < 10;} );
#10
void'(t.randomize() with {addr == 2000; data > 10;});

#10
 t.c1.constraint_mode(0);
void'(t.randomize() with {addr > 101 ; addr < 500;});
#10
data=50;
t.c1.constraint_mode(1);
void'(t.randomize() with { data <= local::data +10 } ) ;
end
```

Uniqueness constraints

- A group of variables can be constrained using the **unique** constraint so that no two members of the group have the same value after randomization.

```
class A;
rand bit [7:0] a[5];
rand byte addr;
rand byte data;
constraint u {
unique {addr, a[2], a[3], data};
}
endclass
```

```
a[2]=4
a[3]=62
addr=55
data=99
```

- variables **a[2]**, a[3], **addr**, and **data** will all contain different values after randomization.

Uniqueness constraints

```
class A;
rand bit 31:0] arr[5];
rand bit [7:0] dyn[];
constraint u {
unique { arr };
unique { dyn };
dyn.size() inside {[4:10]};
}
endclass
```

```
arr[0]=23
arr[1]=8
arr[2]=4
arr[3]=62
arr[4]=46
```

```
dyn[0]=46
dyn[1]=224
dyn[2]=8
dyn[3]=125
dyn[4]=79
```

- All **elements** of **arr** will contain unique value after randomization.
- All **elements** of **dyn** will contain unique value after randomization.

Uniqueness constraints

```
class A;
rand bit 31:0] arr[5];
rand bit [7:0] dyn[];
constraint u {
unique { arr , dyn };
dyn.size() inside {[4:10]};
}
endclass
```

```
arr[0]=23
arr[1]=8
arr[2]=4
arr[3]=62
arr[4]=46
```

```
dyn[0]=22
dyn[1]=87
dyn[2]=220
dyn[3]=125
dyn[4]=5
```

- *All elements of **arr** **and** **dyn** will contain **unique value after randomization**.*

scope randomize function : std::randomize()

- The **scope randomize function**, **std::randomize**(), enables users to randomize data in the current scope without the need to define a class or instantiate a class object.

```
bit [15:0] addr;
bit [31:0] data;
bit success, rd,wr;
```

```
addr=$urandom;
data=$urandom;
wr=$urandom;
```

```
initial begin
success = std::randomize( addr, data, wr );
end
```

std::randomize() with

- Allows users to specify random constraints to be applied to the local scope variables.

```
int a, b, c;

initial begin

success = std::randomize( a, b ) with { a < b ;};

success = std::randomize( a, b, c ) with { (b – a) > c ; };
end
```

Hierarchical randomization

```
class A;
rand bit [3:0] addr;
endclass

class B;
rand bit [7:0] data;
  rand A a1;

function new();
a1=new;
endfunction
endclass
```

```
B b1;
initial begin
b1=new;
void' (b1.randomize());
void' (b1.a1.randomize());
end
```

```
B b1;
initial begin
b1=new;
void' (b1.randomize());
end
```

External constraint blocks

```
class packet;

rand bit [7:0] addr;
rand bit [31:0] data;

extern constraint valid_c ;
endclass

constraint packet::valid_c {
addr inside {[0:15]};
data inside {[100:500]};
}
```

Soft constraints

```
class A ;
rand bit [3:0] x;

constraint c1 {
  soft x==4;      P1
  soft x > 5;     P2
}

constraint c2 {
soft x < 5;       P3
}
endclass

program test;
   A a1;
initial begin
a1=new;
void'(a1.randomize());
$display("a1.x=%0d",a1.x);
end
endprogram
```

Regular (hard) constraints must always be satisfied; they are never discarded.

Soft constraints are discarded when they are contradicted by other hard constraints.

soft constraints may be overridden by hard constraints.

soft constraints may be overridden by other higher-priority soft constraints.

Specific priority shall be associated with every soft constraint.

The soft priorities are designed such that the last constraint specified by the user will prevail.

Constraint inheritance

A derived class shall inherit all constraints from its superclass.

```
class base;

  rand bit [3:0] k;

  constraint valid1 {

    k inside {[2:15]};

}
endclass

class derived extends base;

  constraint valid2 {

     k inside {[5:9]};

}
endclass
```

derived:

base:
k
valid1

valid2

```
program test;
  base b;
  derived d;

initial begin
d=new;
void'(d.randomize());

b=d;
void'(b.randomize());

end

endprogram
```

Constraint inheritance

```
class base;

 rand bit [3:0] k;

 constraint  valid    {

   k inside {[2:4]};

}
endclass

class derived extends base;

 constraint  valid   {

    k inside {[5:9]};

}
endclass
```

derived:

base:
k
valid

valid

```
program test;
  base b;
  derived d;

initial begin
d=new;
void'(d.randomize());

b=d;
void'(b.randomize());

end

endprogram
```

Any constraint in a ***derived class***
having the same name as a constraint in its superclass
shall replace the inherited constraint of that name.

Constraint inheritance

- ***A derived class shall inherit all constraints from its superclass***.
- Any constraint in a **derived class having the same name as a constraint in its superclass** ***shall replace the inherited constraint of that name.***
- Any constraint in a derived class that ***does not have the same name*** as a constraint in the superclass shall be ***an additional constraint.***

pre_randomize()/post_randomize()

Packet:
data = 0
prev_data = 0
max = 0

```
program test;

packet pkt;

initial begin
 pkt=new;

repeat(3) begin

void'(pkt.randomize());

$display("After randomize: pkt.data=%d",pkt.data);

end

end
endprogram
```

Data=6
Data=9
Data=16

```
class packet;
rand bit [7:0]  data;
bit [7:0] prev_data;
bit [7:0] max;

constraint valid {

data inside { [5:max] };

data  !=  prev_data;
}

function void pre_randomize();

    max=$urandom_range(10,20);

endfunction

function void post_randomize();

   prev_data = data;

endfunction

endclass
```

post_randomize()

```
program test;

packet pkt;

initial begin
 pkt=new;

repeat(2) begin

void'(pkt.randomize());

end

end
endprogram
```

```
class packet ;
rand bit [7:0] sa,da;
bit [31:0] len,crc;
rand bit [7:0] payload[];

bit [7:0] inp_stream[$];

constraint valid {
sa inside {[1:4]};
da inside {[1:4]};
payload.size() inside {[2:1900]};
foreach(payload[i])
payload[i] inside {[0:255]};
}

function void post_randomize();
len = payload.size() + 1+1+4+4;
crc = payload.sum();
this.pack(inp_stream);
endfunction

function void pack(ref bit [7:0] q_inp[$]);
  q_inp = {<< 8 {payload,crc,len,da,sa } };
endfunction
endclass
```