# RANDOMIZATION

# AND

# CONSTRAINTS

# Randomization in Verilog and drawbacks

- In Verilog, a system task **$random** for generating random integer values
- Returns **32-bit** random value.
- Returns signed random values
- This is not good for object randomization (here object means class based).
- 100% Constraint randomization is not possible in Verilog
- To help with class-based objects to be randomized, System Verilog supports rand variables and **randomize()** method

**Example: In Verilog**

```
 module tb;
 reg [31:0] address;
        initial begin
            repeat(5) begin
                address=$random();
                $display (“address=%d”,address);
                end  end
endmodule
```

**O/p:**

```
address = 2223298057;
address = 1189058957;
address = 15983361;
address = 512609597;
address = 2097015289;
```

# Differences

## Verilog

Used for design entry

Low  level verification

## System Verilog

Constrained verification

Object oriented randomization

Flexibility to control dynamically

Makes the programming closer to specification

**For Example, Requirement range between -9 to 9**

## Verilog randomization

```
module tb;
integer  address;
initial begin
repeat(5) begin
address= $random %10 ;
$display(“address=%d”,address);
end
end
endmodule
```

O/p:

```
address = 8;
address = 9;
address = 4;
address = 7;
address = 9;
```

**Not as expected**

## SV  CRV

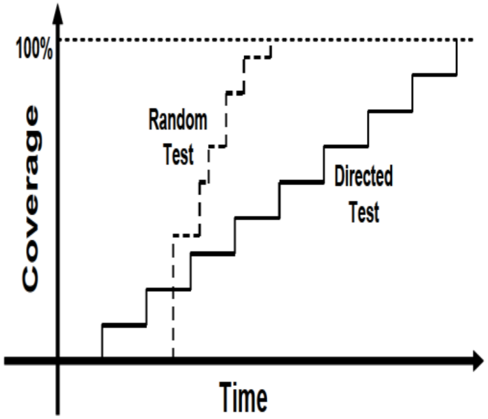
```
class abc;
 rand integer address;
constraint range1{address inside {[-9:9]};}
Endclass
Module tb;
 abc obj=new;
initial begin
repeat(5) begin
Obj.randomize();
$display (“address=%d”,address);
end
end
endmodule
```

As expected

O/P:

```
address = 7;
address = 2;
address = -3;
address = -5;
address = -9;
```

# Why  Randomization?



**Directed and random stimulus generation:**

- Random stimulus is crucial for exercising complex designs
- A **directed test finds** the bugs you expect to be in the design
- A **random test** can find **bugs** you never expected.
- **System Verilog** allows object-oriented ways of random stimulus generation.
- Random test behavior depends upon the variable
- **Randomization** allows 2-state and 4-state types, though randomization only works with 2-state values.
- Can have random integers, bit vectors, etc. and can’t have a random string
- **Constraint stimulus** is useful  for **random** values to reach specification.
- Otherwise, takes too long to generate interesting stimulus or the stimulus might contain illegal values.

# rand and randc

- For randomization, variables declared as rand or randc inside class are only randomized
- randc variables has the new random values unless it repeats all the possible values

Ex:

```
class Packet;
  rand bit [1:0] addr;
endclass
module tb;
 Packet Pkt=new;
initial begin
repeat(4)
 begin
  Pkt.randomize();
   $display("Pkt.addr=%d",Pkt.addr);
end
end
endmodule
```

O/p:

```
Pkt.addr=3
Pkt.addr=0
Pkt.addr=3
Pkt.addr=3
```

Ex:

```
class Packet;
  randc bit [1:0] addr;
endclass
module tb;
 Packet Pkt=new;
initial begin
repeat(4)
 begin
  Pkt.randomize();
   $display("Pkt.addr=%d",Pkt.addr);
end
end
endmodule
```

O/p:

```
Pkt.addr=3
Pkt.addr=2
Pkt.addr=0
Pkt.addr=1
```

# randomize()

- The randomize() method is a **virtual function** that generates random values for all the active random variables in the object with respect to active constraints.
- The randomize() method returns ‘1’, if it successfully randomizes all the random variables inside an objects , otherwise it returns ‘0’.
- In order to randomize the object variables, need to call randomize() method

Syntax: object_handle.randomize();

**Example**:

```
class Packet;
  rand bit [1:0] addr;
  randc bit [3:0] data;
endclass
module tb;
 Packet Pkt=new;
initial begin
repeat(4)
 begin
  Pkt.randomize();
   $display("Pkt.addr=%d, Pkt.data=%d  ",Pkt.addr,Pkt.data);
end
end
endmodule
```

o/p:

```
Pkt.addr=3, Pkt.data=11
Pkt.addr=3, Pkt.data=15
Pkt.addr=3, Pkt.data= 1
Pkt.addr=1, Pkt.data= 2
```

# Random and Non-Random Variables

```
class Packet;
  rand bit [1:0] addr;
  randc bit [3:0] data;
  int read;
endclass
module tb;
 Packet Pkt=new;
initial begin
repeat(4)
 begin
  Pkt.randomize();
   $display("Pkt.addr=%d, Pkt.data=%d ,Pkt.Read=%d ",Pkt.addr,Pkt.data,Pkt.read);
end
end
endmodule
```

**O/P:**

```
Pkt.addr=3, Pkt.data=11 ,Pkt.Read=          0
Pkt.addr=3, Pkt.data=15 ,Pkt.Read=          0
Pkt.addr=3, Pkt.data= 1 ,Pkt.Read=          0
Pkt.addr=1, Pkt.data= 2 ,Pkt.Read=          0
```

# Constraints block

- Contains declarative statements which restrict the range of variable or defines the relation between variables.
- Lets users to build generic, reusable objects that can be extended or more constrained later.
- Constraint solver can only support 2 state values.
- If a 4 state variable is used, solver treats them as 2 state variable.
- Constraint solver fails only if there is no solution which satisfies all the constraints.
- Constraint block can also have non-random variables, but at least one random variable is needed for randomization.
- Allows inheritance, hierarchical constraints, controlling the constraints of specific object

# Contd.. Example

```
class Packet;
  rand integer unsigned addr;

  constraint addr_range {
                addr < 100 ;
                                    }
endclass

Packet Pkt;
module tb;
initial
begin
     repeat(3)
     begin
         Pkt=new();
        Pkt.randomize();
        $display("------------------------");
        $display("Packet Var=  %d",Pkt.addr);
        $display("------------------------");
      end
  end
 endmodule
```

Result:

```
------------------------
Packet Var= 38
------------------------
------------------------
Packet Var= 10
------------------------
------------------------
Packet Var= 23
------------------------
```

# Set membership : Inside operator

- Variables will only get randomized with the values mentioned using inside operator
- Values within the inside block can be variable, constant or range.
- A set membership is a list of expressions or a range.
- This operator searches for the existences of the value in the specified expression or range and returns 1 if it is existing.
- If you want to define a range which is outside the set, use negation.

**Example :**

```
rand bit [3:0] start_addr;
rand bit [3:0] end_addr;
rand bit [3:0] addr;

   constraint addr_range{addr inside {1,3,[5:10],12,[13:15]}; }

   constraint addr_range { addr inside {[start_addr:end_addr]}; }
```

# Example:

```
class Packet;
  rand integer unsigned addr;
  constraint addr_range {
              addr inside {[0:15]};
       }
endclass
Packet Pkt;
module tb;
initial
begin
    repeat(3)
    begin
        Pkt=new();
        Pkt.randomize();
        $display("------------------------");
        $display("Packet Var=  %d",Pkt.addr);
        $display("------------------------");
   end
end
endmodule
```

Result:

```
------------------------
Packet Var= 1
------------------------
------------------------
Packet Var= 3
------------------------
------------------------
Packet Var= 8
------------------------
```

```
constraint addr_range {
                    addr inside {0,4,6,8,10
                             };
```

**Result:**

```
------------------------
Packet Var= 0
------------------------
------------------------
Packet Var= 6
------------------------
------------------------
Packet Var= 4
------------------------
```

```
constraint addr_range {
        addr inside {[0:4],6,[8:10]};
                              }
```

**Result:**

```
------------------------
Packet Var= 3
------------------------
------------------------
Packet Var= 2
------------------------
------------------------
Packet Var= 8
------------------------
```

```
constraint addr_range {
              addr inside {[0:20]};
              addr>0;
              addr<100;
              }
```

**Result:**

```
------------------------
Packet Var= 4
------------------------
------------------------
Packet Var= 11
------------------------
------------------------
Packet Var= 9
------------------------
```

# Constraints solver failure

```
constraint addr_range {
                    addr inside {100,120};
              addr>0;
              addr<100;
              }
```

```
==========================================================

Solver failed when solving following set of constraints
rand bit[31:0] addr; // rand_mode = ON
constraint addr_range // (from this) (constraint_mode = ON)
(testbench.sv:5)
{
(addr inside {100, 120});
(addr < 100);
}
```

# Distribution(dist)

- By := or :/ operator, some values can be allocated more often to a random variable.
- It takes a list of values and weights depending upon := and :/
- The values and weights can be
  a) Constants or variables
  b) Single or a range
  c) Default weight of an unspecified value is 1

## Differences between := and :/

:=

- Assigns the specified weight to the item
- If the item is a range, specified weight to every value in the range.

Ex: addr **dist** { 2 := 5,[10:12] := 8 };

```
addr = 2 , //weight 5
addr = 10, // weight 8
addr = 11, //weight 8
addr = 12,// weight 8
```

:/

- Assigns weight to the item
- If the item is a range, specfied weight /n to every value in the range

**Note:** where n is number of values in the range.

Ex: addr **dist** { 2 :/ 5,[10:12] :/ 8 };

```
addr = 2 , //weight 5
addr = 10, //weight 8/3
addr = 11, //weight 8/3
addr = 12,// weight 8/3
```

```
class Packet;
  rand integer unsigned addr;
constraint addr_range {
  addr dist {2:=20, 100:=40};
                    }

endclass

Packet Pkt;
module tb;
initial
begin
  repeat(5)
  begin
        Pkt=new();
        Pkt.randomize();

        $display("Packet Var=  %d",Pkt.addr);
  end
end
endmodule
```

```
O/P:
Packet Var= 100
Packet Var= 100
Packet Var= 2
Packet Var= 100
Packet Var= 100
```

```
constraint addr_range {
  addr dist {[2:4]:=20, [5:8]:/40};
                    }
```

```
O/P:
Packet Var= 6
Packet Var= 8
Packet Var= 5
Packet Var= 7
Packet Var= 3
Packet Var= 3
```

# Foreach iterative constraints

```
class Packet;
 rand bit[31:0] arr_addr[10];
constraint addr_range {
               foreach(arr_addr[index])
               {
                 arr_addr[index] inside {1,2,10,8};
               }
    }

endclass

Packet Pkt;
module tb;
initial
begin
        Pkt=new();
        Pkt.randomize();
        foreach(Pkt.arr_addr[loop])
          $display("Packet Var[%0d]=
%0d",loop,Pkt.arr_addr[loop]);
end
endmodule
```

```
Packet Var[0]= 10
Packet Var[1]= 1
Packet Var[2]= 8
Packet Var[3]= 1
Packet Var[4]= 10
Packet Var[5]= 8
Packet Var[6]= 1
Packet Var[7]= 8
Packet Var[8]= 2
Packet Var[9]= 10
```

```
rand bit[31:0] arr_addr[];
constraint addr_range {
               foreach(arr_addr[index])
               {
                 arr_addr[index] inside {1,2,10,8};
               }
               arr_addr.size() == 4;
          }
```

```
Packet Var[0]= 10
Packet Var[1]= 1
Packet Var[2]= 8
Packet Var[3]= 1
```

```
rand bit[31:0] arr_addr[$];
constraint addr_range {
               foreach(arr_addr[index])
               {
                 arr_addr[index] inside {1,2,10,8};
               }
               arr_addr.size() == 5;
               arr_addr.sum() == 8;
     }
```

```
Packet Var[0]= 2
Packet Var[1]= 2
Packet Var[2]= 1
Packet Var[3]= 2
Packet Var[4]= 1
```

# If-else and Implication(->)

- Constraints provide two constructs for declaring conditional (predicated) relations: implication and **if-else**.
- **implication :**The Boolean equivalent of the implication operator a **->** b is (!a || b). This states that if the expression is true, then random numbers generated are constrained by the constraint (or constraint set). Otherwise, the random numbers generated are unconstrained

# If-else

```
class Packet;
  rand bit[3:0] addr;
  rand   bit[3:0] data;

  constraint cnst{
     data inside {1,3,2,4};
    if(addr > 4) data == 5;
    else          data == 2;
  }
endclass

Packet Pkt;
module tb;
initial
begin
  repeat(3)
        begin
              Pkt=new();
              Pkt.randomize() ;//with {addr.size()<5;};
              $display("------------------------");
          $display("Packet Var=  %0p",Pkt);
              $display("------------------------");
          end
end
endmodule
```

**Result:**

```
------------------------
Packet Var= '{addr:'h0, data:'h2}
------------------------
------------------------
Packet Var= '{addr:'h2, data:'h2}
------------------------
------------------------
Packet Var= '{addr:'h2, data:'h2}
------------------------
```

```
constraint cnst{    data inside {1,3,2,4,5};
                    if(addr>4) data == 5;
                    else          data == 2;
                }
```

**Result:**

```
------------------------
Packet Var= '{addr:'h4, data:'h2}
------------------------
------------------------
Packet Var= '{addr:'h6, data:'h5}
------------------------
------------------------
Packet Var= '{addr:'h2, data:'h2}
```

# Implication

```
class Packet;
  rand bit[3:0] data;
  rand enum{LOW,HIGH}Write;

constraint cnst_wr_data {
            (Write==LOW) ->  data < 8;     /*  if(Write==LOW)   data < 8
                                                 else                  data >= 8*/
}


endclass

Packet Pkt;
module tb;
initial
begin
        Pkt=new();
        repeat(5) begin
           Pkt.randomize();
          $display("Packet=%p",Pkt);
        end

end
endmodule
```

**Result:**

```
Packet='{data:'h7, Write:HIGH}
Packet='{data:'h2, Write:LOW}
Packet='{data:'h1, Write:HIGH}
Packet='{data:'h6, Write:LOW}
Packet='{data:'h6, Write:LOW}
```

# Unique

- All members of the group of variables so specified (that is, any scalar variables, and all leaf elements of any arrays or slices) shall be of equivalent type.
- No **`randc`** variable shall appear in the group.

```
class Packet;
  rand bit[3:0] data;
  rand bit[3:0] addr;

constraint cnst_wr_data {
  unique {addr,data};
    }
endclass

Packet Pkt;
module tb;
initial
begin
        Pkt=new();
        repeat(5) begin
           Pkt.randomize();
          $display("Packet=%p",Pkt);
        end
end
endmodule
```

```
Packet='{data:'h7, addr:'h8}
Packet='{data:'h4, addr:'h0}
Packet='{data:'h1, addr:'he}
Packet='{data:'hd, addr:'h6}
Packet='{data:'hd, addr:'h5}
```

# Inline Constraints - with

Inline constraints is done using **with** keyword

Adds constraints when **randomize()** method is called

```
class Packet;
  rand bit[3:0] data;
  rand bit[3:0] addr;

constraint cnst_wr_data {
                        addr inside {[1:10]};
                                 }
endclass

Packet Pkt;
module tb;
initial
begin
      Pkt=new();
      repeat(5) begin
        Pkt.randomize() with {addr>3;data<3;};
        $display("Packet=%p",Pkt);
      end

end
endmodule
```

```
Packet='{data:'h1, addr:'h5}
Packet='{data:'h2, addr:'h9}
Packet='{data:'h2, addr:'ha}
Packet='{data:'h1, addr:'h4}
Packet='{data:'h0, addr:'h8}
```

# soft

```
class Packet;
  rand bit[9:0] addr;

constraint cnst_wr_data {
  soft addr ==2 ;
    }

endclass

Packet Pkt;
module tb;
initial
begin
        Pkt=new();
        Pkt.randomize();
  $display("Packet addr=%d",Pkt.addr);
        #1;
  Pkt.randomize() with {addr>10;};
  $display("Packet addr=%d",Pkt.addr);


end
endmodule
```

```
Packet addr= 2
Packet addr= 182
```

# Bidirectional Constraints

- System Verilog constraints solved *bidirectionally*, which means constraints on all random variables will be solved parallel.
- Constraint solver will consider all the constraints to choose values to all the random variable, because constrained value of one variable may depends on the value of other variable, which may be again constrained.

# Disabling random variables

- The random nature of variables declared as rand or randc can be turned **on** or **off** dynamically
- The rand_mode() method can be used to disable the randomization of variable declared with rand/randc.
- By default rand_mode value for all the random variables will be 1.
- after setting rand_mode(0) to any random variable, it will get randomized only after rand_mode(1).
- The rand_mode() method is built-in and cannot be overridden

Syntax:

variable _name. rand_mode(0);

## Example:

```
module sv_random();
class Packet;

  randc bit [3:0] Wdata;

  constraint cnst{
     Wdata < 6;
  }
endclass

Packet obj=new();
  initial
    begin
          obj.randomize();
          $display("After randomize Wdata=%d",obj.Wdata);
          obj.rand_mode(0);
          $display("After rand_mode_off Wdata=%d",obj.Wdata);

      obj.randomize();// with {Wdata <2;};
          $display(" randmized when rand_mode_0 Wdata=%d",obj.Wdata);
          obj.rand_mode(1);

      obj.randomize(); //with {Wdata <2;};
          $display(" randmized when rand_mode_1 Wdata=%d",obj.Wdata);

   end
endmodule
```

```
After randomize Wdata= 4
After rand_mode_off Wdata= 4
randmized when rand_mode_0 Wdata= 4
randmized when rand_mode_1 Wdata= 0
```

O/P

```
After randomize Wdata= 4
After rand_mode_off Wdata= 4
Solver failed when solving following set of constraints


bit[3:0] Wdata = 4'h4;

constraint cnst // (from this) (constraint_mode = ON) (testbench.sv:7)
{
(Wdata < 4'h6);
}
constraint WITH_CONSTRAINT // (from this) (constraint_mode = ON) (testbench.sv:20)
{
(Wdata < 4'h2);
}


=============================================================

Please check the inconsistent constraints being printed above and rewrite
them.

randmized when rand_mode_0 Wdata= 4
randmized when rand_mode_1 Wdata= 1
```

# Constraint mode

- To change the status of a Constraint block, built in constraint_mode() method is used.
- By default all the constraint blocks are active(constaint_mode(1))
- We can pass 0 as argument to constarint_mode() to remove constraints on a class object or variable

```
class Packet;
  rand integer unsigned  Var1,Var2;
  constraint Var_1 { Var1 == 20;}
  constraint Var_2 { Var2 == 20;}
endclass

module sv_random;
Packet obj ;
initial begin
  obj=new();
     begin
  obj.randomize();
      $display("constraint_mode ON Var1:%d Var2 : %d ",obj.Var1,obj.Var2);
  obj.constraint_mode(0);
  obj.randomize();
      $display("constraint_mode OFF Var1 : %d Var2 : %d ",obj.Var1,obj.Var2);
           obj.constraint_mode(1);
  obj.randomize();
      $display("constraint_mode ON Var1 : %d Var2 : %d ",obj.Var1,obj.Var2);
  obj.constraint_mode(0);
end
end
endmodule
```

**Result:**

```
constraint_mode ON Var1:        20 Var2 :         20
constraint_mode OFF Var1 :  924167702 Var2 : 2928684018
constraint_mode ON Var1 :         20 Var2 :         20
----------------------------------------------------------
```

# pre_randomize & post_Randomize Functions

- Every class contains **pre_randomize**() and **post_randomize**() methods, which are automatically called by **randomize**() before and after computing new random values.
- When **randomize**() is called, it first invokes the **pre_randomize**(), then **randomize**() finally for the user to perform operations such as setting initial values and performing if the randomization is successful only **post_randomize**() is invoked.
- The **pre_randomize**() and **post_randomize**() methods are not virtual. However, because they are automatically called by the **randomize**() method, which is virtual, they appear to behave as virtual methods.
- Users can override the **pre_randomize**() in any class to perform initialization and set values before the object is randomized
- Users can override the **post_randomize**() in any class to perform cleanup, and check post-conditions after the object is randomized.

# Contd..

- If the class is a derived class and no user-defined implementation of **pre_randomize**() exists, then **pre_randomize**() will automatically invoke **super.pre_randomize**().
- If the class is a derived class and no user-defined implementation of **post_randomize**() exists, then **post_randomize**() will automatically calls **super.post_randomize**()
- If these methods are overridden, they shall call their associated base class methods; otherwise, their pre- and post-randomization processing steps shall be skipped.

Example:-

```
program pre_post_15;
  class simple;
      function void  pre_randomize;
        $display(" PRE_RANDOMIZATION ");
      endfunction
      function void  post_randomize;
        $display(" POST_RANDOMIZATION ");
      endfunction
  endclass
  simple obj = new();
  initial
    obj.randomize();                o/p:   PRE_RANDOMIZATION
endprogram                                  POST_RANDOMIZATION
```

# Example

```
module sv_random;
         class Packet;
      rand bit [3:0]Wdata;
      rand bit [2:0]addr;

         function void pre_randomize();
      $display("\npre_randomisation");
          Wdata=2;addr=7;
      $display("\tWdata =%d,\taddr= %d",Wdata,addr);
          endfunction

         function void post_randomize();
      $display("\npost_randomisation");
      $display("\tWdata=%d\t,\taddr=%d",Wdata,addr);
         endfunction

         endclass
initial begin
           Packet obj =new();
  repeat(3)
                  obj.randomize();
end endmodule
```

o/p:

```
pre_randomisation
Wdata = 2, addr= 7

post_randomisation
Wdata= 7 , addr=0

pre_randomisation
Wdata = 2, addr= 7

post_randomisation
Wdata=11 , addr=3

pre_randomisation
Wdata = 2, addr= 7

post_randomisation
Wdata= 9 , addr=7
```

# Contd..

**Behavior of randomization methods**

- 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 called.
- The randomize() method is built-in and cannot be overridden.
- The randomize() method implements object random stability. An object can be seeded by calling its srandom() method
- The built-in methods pre_randomize() and post_randomize() are functions and cannot block.

# randcase

randcase is a case statement that randomly selects one of its branches.

The randcase item expressions are non-negative integral values that constitute the branch weights.

**Example :**

```
randcase
    3 : x = 1;
    1 : x = 2;
    4 : x = 3;
endcase
```

## Randsequence:

randsequence generator is useful for randomly generating sequences of stimulus.

By randomizing a packet:

- it will generate most unlikely scenarios which are not interested .
- These type of sequence of unlike scenarios can be avoided using randsequence

# Example:

randsequence:

```
module tb;
initial
begin
  repeat(3)
        begin
                randsequence( main )
                        main :  one two three ;
                        one : {$write("one");};
                        two : {$write(" two");};
                        three: {$display(" three");};
                endsequence
   end
end
endmodule
```

**O/P**

```
one two three
one two three
one two three
```

randcase:

```
initial
begin
repeat(5)
begin
randcase
3:$display("moschip with highest weight");
1:$display("moschip with least weight");
2:$display("moschip with medium weight ");
endcase
end
```

**O/P:**

```
moschip with highest weight
moschip with medium weight
moschip with highest weight
moschip with medium weight
moschip with medium weight
moschip with medium weight
moschip with highest weight
moschip with highest weight
moschip with highest weight
moschip with least weight
```

# Randomizing classes

Similar to struct, the same can be achieved using class by calling the randomize() function on the object, which is created by using class.

**Example:**

```
module class_rand;
  class Packet;
    rand bit[2:0] Addr;
endclass

  class my_Packet;
    rand Packet Pkt
    randc bit[2:0] Wdata;

          function new();
               Pkt = new();
           endfunction
  endclass

  my_Packet M_Pkt = new();
   initial begin
    repeat(4)
      M_Pkt.randomize() with {Wdata > 5;Addr < 8};
        $display(" Var1 : %0d Var2 : %0d",M_Pkt.Pkt.Addr,M_Pkt.Wdata);
   end
endmodule
```

**O/P:**

```
Addr : 0 Wdata : 6
Addr : 1 Wdata : 3
Addr : 3 Wdata : 0
Addr : 1 Wdata : 7
```

bit[2:0] Addr;

```
Addr : 0 Wdata : 6
Addr : 0 Wdata : 7
Addr : 0 Wdata : 3
Addr : 0 Wdata : 4
```

bit[2:0] Wdata;

```
Addr : 7 Wdata : 0
Addr : 0 Wdata : 0
Addr : 3 Wdata : 0
Addr : 3 Wdata : 0
```

# Constraint Inheritance

- Constraints also will get inherited from parent class to child class.
- Parent class constraint blocks are overridden in child class constraints of same variable

```
class Packet;
    rand integer Var;
  constraint Var_range {
                    Var > 150 ;Var<200;
                        }
 endclass

  class M_Packet extends Packet;
constraint Var_range {
                     Var < 100 ;
                     Var > 0 ;
                           }
   endclass
```

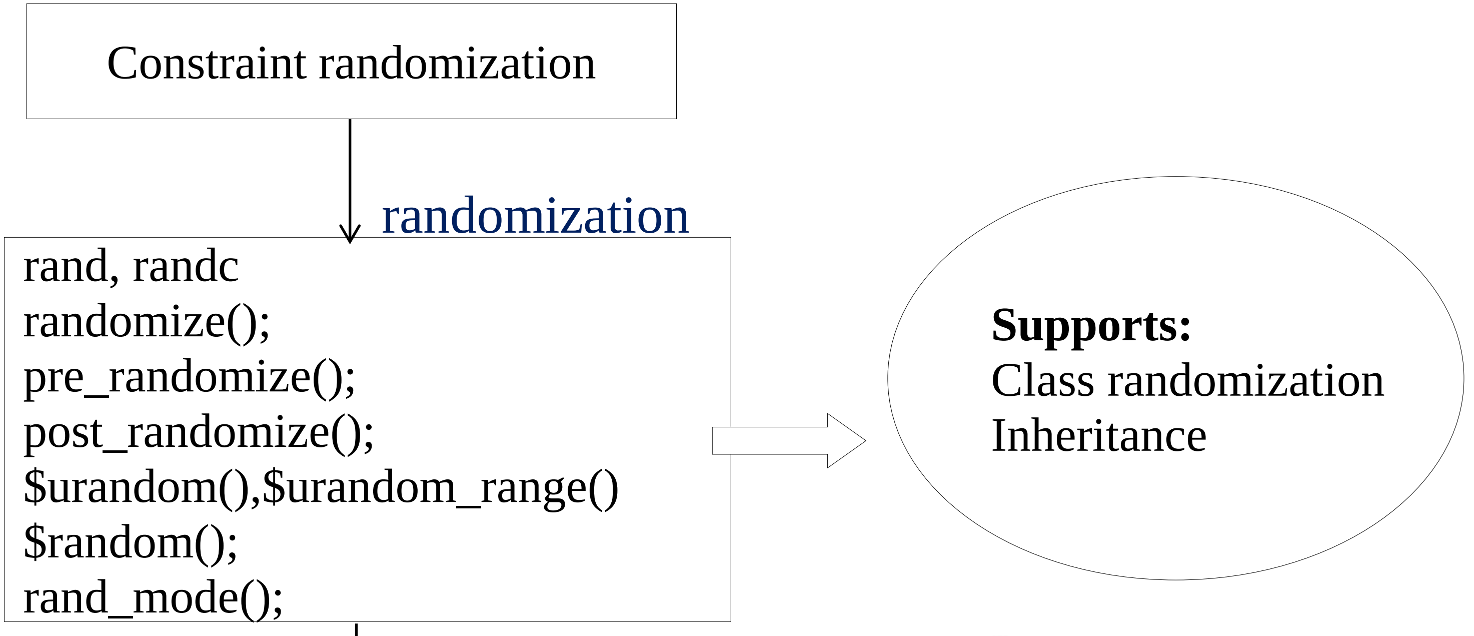
```
module SV_random;
initial begin
Packet Pkt= new();
M_Packet M_Pkt= new();

   M_Pkt.randomize();
   $display("Var = %0d ",M_Pkt.Var);
   Pkt .randomize();
   $display("Var = %0d ",Pkt.Var);
   Pkt = M_Pkt;//handle assignment
   Pkt .randomize();
   $display("Var = %0d ",Pkt.Var);
end
endmodule
```

**O/P**

```
Var = 40
Var = 189
Var = 37
```

# Constraint Randomization:



Constraint randomization

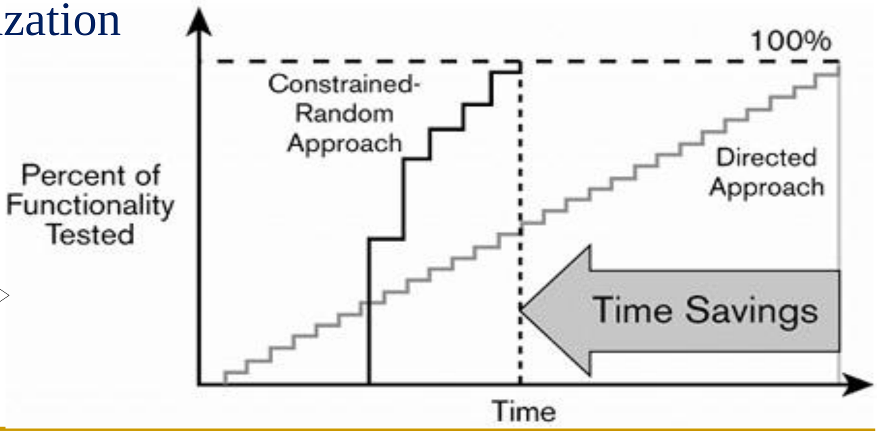
randomization

rand, randc
randomize();
pre_randomize();
post_randomize();
$urandom(),$urandom_range()
$random();
rand_mode();

**Supports:**
Class randomization
Inheritance

Constraints + randomization

foreach
set membership
inline constraint
conditional constraints
distribution constraints
constraint_mode();
variable ordering


Constrained-Random Approach
Percent of Functionality Tested
Directed Approach
Time Savings
Time