ASIC Verification

Randomization

Fall 2011
Meeta Yadav
Topics

• Why use randomization
• Randomization options
• Randomization of objects
• Class constraints and distributions
• In-line constraints and distributions
Randomization Overview

Why Randomize?

- As designs grow it becomes more difficult to test their features through directed test cases.
- Directed testcases checks specific features of a design and can only detect anticipated bugs.
- Directed testcase approach is a time consuming exercise.
- Directed testscases are unable to detect devious bugs.
- Directed tests detect only anticipated bugs in the design.

Solution is Constrained Random Testcases (CRT) which randomize the input stimulus.

Random testing detects the bugs you did not expect in your design.
What to Randomize?

- **Device configurations**
  - Try different device configurations
  - For instance for a switch verification try different configurations of input and output ports

- **Environment configurations**
  - Randomize the entire environment that the device operates in
    - The number of objects
    - Object configurations
  - For instance randomize the number of PCI busses connected to a particular device
What to Randomize?

• **Primary Input Data**
  - Randomize input data
  - Constrain the inputs to be within valid limits

• **Encapsulated input data**
  - If the data is encapsulated, the different layers of encapsulation should be randomized and the device should be tested with different input configurations
  - The encapsulated data should also be randomized
What to Randomize?

• **Protocol exceptions, errors and violations**
  ◆ Verify how system handles errors
  ◆ Anticipate error cases and inject into the system to ensure that the design gracefully handles them
  ◆ Inject errors randomly

• **Delays**
  ◆ Use random legal delays specified by the protocols
Randomization in SystemVerilog

• Overview
  ◆ Randomization enables users to automatically generate random input stimulus for functional verification
  ◆ SystemVerilog enables user to specify random constrained (legal) values
  ◆ Random constraints should be specified using OOP
Randomization in SystemVerilog

- **rand** keyword
  - Random variables are declared with the **rand** keyword
  - Their values are uniformly distributed over the specified range
  - If unconstrained the variable shall assign any value in the specified range with equal probability

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

- 8-bit unsigned integer with the range 0 to 255
- If unconstrained `y` is assigned any value in the range 0 to 255 with equal probability
- The probability of a value occurring on simultaneous calls to randomize is $1/256$
Randomization in SystemVerilog

- **`randc` keyword**
  - Random cyclic variables are declared with the `randc` keyword
  - They cycle through all the values in a random permutation of their declared range
  - Can only be of the type `bit` or `enum`

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

- `randc` randomly iterates over all the values in the range and no value is repeated within an iteration
  - When the iteration finishes, a new iteration automatically starts

  **y can take values 0,1,2,3**

  - Initial permutation: 0 -> 3 -> 2 -> 1
  - Next permutation: 2 -> 1 -> 3 -> 0
  - Next permutation: 2 -> 0 -> 1 -> 3
### Randomization in SystemVerilog

- **Simple class with random variables**
  - The **Bus** class models a simplified bus with two random variables: `addr` and `data`
  - The **range1** constraint specifies the limits for the values of `addr`
    - Ensure non conflicting and legal constraints

```systemverilog
class Bus
    rand bit[15:0] addr;
    randc bit[31:0] data;
    constraint range1 {addr>1024; addr<16384;}
endclass
```

**Example: Simple class with random variables**
- random variable
- random cyclic: the random solver will not repeat a permutation of values
- constraining the random variables to values between 1024 and 16384
Randomization in SystemVerilog

- **randomize() function**
  - **Calling randomize()** causes new values to be selected for all of the random variables in an object
    - The random values obey the constraints
  - **randomize function** returns a 1 on success and 0 on failure
  - **Unconstrained variables** are assigned any values in their declared range

A bus object is created and randomized 50 times. The result of each randomization is checked.

```systemverilog
Bus bus=new;
repeat (50) begin
  if(bus.randomize()==1)
    $display(bus.addr, bus.data);
  else
    $display ("Randomization failed");
end
```

*Example: randomize() function example*
Randomization in SystemVerilog

• The constraint solver
  ◆ Solves constraint expressions
  ◆ The same seed results in the same random values
    ▶ Use a different seed to generate different set of random values
  ◆ The solver is specific to the simulation vendor
Constraint Details

- **Constraint Blocks**
  - Values of random variables are determined using constraint expressions that are declared using constraint blocks
    - Smart stimulus tests relationships between variables
  - They are class members like tasks, functions and variables
  - Constraint declaration
    - **Constraint identifier**: is the name of the constraint block.
    - **Constraint block**: is a list of expression statements that restrict the range of variable or define relations between variables

In the example below randomize() fails sometimes and succeeds sometimes, why?

**Example: Simple constraint example**

```plaintext
class Bus
  bit[15:0] addr;
  randc bit[31:0] data;
  constraint range1 {addr>1024; addr<16384;}
endclass
```

©2011, Dr. Meeta Yadav, www.asicverification.org
Constraint Details

- Simple expressions
  - Constraint variables have to be in a fixed order
    - There can be only one relational operator (<, <=, ==, >= or >)
    - For multiple variables use multiple expressions

Declare a constraint to solve: $0 < a < b < c$

Error!!! wrong way

class Bus
rand bit[15:0] a,b,c;
constraint cl { 0 < a < b < c; }
endclass

right way
Constraint Details: Example

• Set membership Operator: inside
  ◆ If other constraints are absent, all values have an equal probability of being chosen by the inside operator
  ◆ The negated (!) form of inside operator denotes that expression lies outside the set

```verilog
class Bus
  rand bit[15:0] addr;
  randc bit[31:0] data;
  constraint range1 {
    addr inside {{[0:100],[1024:16384]});
    data > 1000;
    data < 10000;
  }
endclass
```

*Example: Constraint example with inside operator*
Constraint Details: Quiz

Declare constraint blocks c1,c2,c3 and c4 so that variable x, a and v get the values shown:

x takes any values described in the constraint_block:
- x is: 3,5,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31,32
- x is: 3,5,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31,32, between y and 2y, z

```plaintext
rand integer x,y,z;
constraint c1
constraint c1
```

- b<=a and a<=c or b<=a<=c

```plaintext
rand integer a,b,c;
constraint c2
```

- c3: v is either 5,10,15,20
- c4: v is never 5,10,15,20

```plaintext
integer fives[0:3] = {5,10,15,20};
rand integer v;
constraint c3
constraint c4
```
Weighted Distribution

• **Distributions:** `dist` operator
  - Used to weigh some values more than the others
  - Can use `:=` or `:/` operator
    - `:=` specifies that the weight has to be the same for every specified value in the range
    - `:/` operator specifies that the weight is to be equally divided between all values. If there are `n` values in a range the weight is `range_weight/n`
  - Values can be a single value or range such as `[lo:hi]`
  - The weights are not percentage and do not have to add up to a 100
  - Cannot be used with a `randc`

```
rand integer a;
constraint c1 {a inside {0,1,1,1,1,1}};
```

Error!!! wrong way

```
rand integer a;
constraint c1 {a dist {0:=1, 1:=5}};
```

right way

`c1 should be 0 once and 1 five times`
Weighted Distribution

• Another example

```
rand int src, dst;
constraint cl {
    src dist {0:=40, [1:3]:=60};
    dst dist {0:=40, [1:3]:/60};
}
```

Example: Weighted random distribution with `dist`

<table>
<thead>
<tr>
<th>src</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>___</td>
</tr>
<tr>
<td>1</td>
<td>___</td>
</tr>
<tr>
<td>2</td>
<td>___</td>
</tr>
<tr>
<td>3</td>
<td>___</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dst</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>___</td>
</tr>
<tr>
<td>1</td>
<td>___</td>
</tr>
<tr>
<td>2</td>
<td>___</td>
</tr>
<tr>
<td>3</td>
<td>___</td>
</tr>
</tbody>
</table>
Weighted Distribution: Quiz

Distributions: \texttt{dist} operator := and :/

- \( x \) is 100, 200 and 300 with a weight of 1, 2 and 5 respectively

- \( x \) is 100 and 300 with a weight of 1 and 5 respectively
- \( x \) is never 200

- \( x \) is 100, 101, 102 with a weight of 1 each
- \( x \) is 200 and 300 with a weight of 2 and 5 respectively

- \( x \) is 100, 101, 102 with a weight of 1/3 each
- \( x \) is 200 and 300 with a weight of 2 and 5 respectively
Bidirectional Constraints

- Bidirectional Constraints
  - Constraint blocks are not procedural but declarative
    - All active at one time

Solve the constraint

\[30 > d > b \Rightarrow c \geq 25\]

```
rand logic [15:0] b, c, d;
constraint c1 {
    b < d;
    c == b;
    d < 30;
    c >= 25;
}
```

Example: Bidirectional constraints

<table>
<thead>
<tr>
<th>Solution</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solutions for Bidirectional Constraints
Conditional Constraints

- Conditional constraint operators
  - Constraint provide two constructs for declaring conditional relations
    - Implication operator ->
    - if...else

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

**Example: if...else example**

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

**Example: Equivalent implication example**

```plaintext
bit [3:0] a, b;
if (a == 0)
    b == 1;
```

**Example: if...else example**

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

**Example: Equivalent implication example**
Solution Probabilities

- Unconstrained
  - Probability of distribution

Example: Class with unconstrained random variables

```verilog
class Unconstrained;
  rand bit x;
  rand bit [1:0] y;
endclass
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1/8</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1/8</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td>1/8</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td>1/8</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>1/8</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1/8</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td>1/8</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td>1/8</td>
</tr>
</tbody>
</table>

There are 8 possible solutions and because there are no constraints each has the same probability.

Solutions for Unconstrained class
Solution Probabilities

- Implication
  - Probability of distribution changes due to the implication operator
  - Implication is bidirectional

```class Imp1;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        (x==0) -> y==0;
    }
endclass
```

Value of y depends on x. When x is 0, y is 0. Hence for x=0, y cannot have any other value and hence the probability of x=0 and y!=0 is 0

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Solutions for **Imp1** class
Solution Probabilities

- Implication and bidirectional constraints
  - Probability of distribution changes due to an addition of a new constraint

```
class Imp_Bid;
  rand bit x;
  rand bit [1:0] y;
  constraint c_xy {
    y>0;
    (x==0)->y==0;
  }
endclass
```

Example: Class with implication and constraint

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

When x is 0, y is 0 but when y is 0 there is no constraint on x. Since implication is bidirectional, if y was forced to a non zero value, x would have to be 1. Hence x can never be 0.

Solutions for Imp_Bid class

©2011, Dr. Meeta Yadav, www.asicverification.org
Solution Probabilities

- **Solve...before**
  - Solve...before does not change the solution space but changes the probability of the results

```plaintext
class SolveBefore;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        (x==0) -> y==0;
        solve y before x;
    }
endclass
```

*Example: Class with implication and solve...before*
Solution Probabilities

```verilog
class SolveBefore;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        (x==0) -> y==0;
        solve y before x;
    }
endclass
```

*Example: Class with implication and solve...before*

<table>
<thead>
<tr>
<th>Solution</th>
<th>y</th>
<th>x</th>
<th>Unconstrained Probability</th>
<th>Solution Possibility</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1/8</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1/8</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>1/8</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1/8</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>0</td>
<td>1/8</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>1</td>
<td>1/8</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>0</td>
<td>1/8</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>H</td>
<td>3</td>
<td>1</td>
<td>1/8</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

*Solutions for solve y before x constraint*
Solution Probabilities

- **Solve...before**
  - Solve...before does not change the solution space but changes the probability of the results

```plaintext
class SolveBefore;
  rand bit x;
  rand bit [1:0] y;
  constraint c_xy {
    (x==0) -> y==0;
    solve y before x;
  }
endclass
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>y</th>
<th>x</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1/8</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1/8</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1/4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>1</td>
<td>1/4</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>3</td>
<td>1</td>
<td>1/4</td>
</tr>
</tbody>
</table>

*Example: Class with implication and solve...before constraints*

*Solutions for solve y before x constraint*
Solution Probabilities

• Solve...before

### Example: Class with implication and `solve x before y`

```plaintext
class Imp4;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        (x==0) -> y==0;
        solve x before y;
    }
endclass
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>1/8</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1/8</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

*Solutions for `solve x before y` constraint*
Constraints

• Iterative constraints
  ◆ Allows arrayed variables to be constrained in a parameterized manner using loop variables

C1 constraints each element of an array to be in the set [2,4,8,16]
C2 constraints each element of an array to be greater than twice its index

```plaintext
class C;
  rand byte A[4];
  constraint C1{ foreach (A[i]) A[i] inside {2,4,8,16};}
  constraint C2{ foreach (A[j]) A[j] > 2*j;}
endclass
```
Disabling Random Variables

• Disabling random variables with `rand_mode()`
  - Can be used to control whether a random variable is active or inactive
    - When a random variable is inactive it implies that the variable was never declared `rand` or `randc`
    - `rand_mode()` method is inbuilt and cannot be overridden

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OFF</td>
<td>Sets the specified variable to inactive so that they are not randomized on subsequent calls to <code>randomize()</code> method</td>
</tr>
<tr>
<td>1</td>
<td>ON</td>
<td>Sets the specified variable to active so that they are randomized on subsequent calls to <code>randomize()</code> method</td>
</tr>
</tbody>
</table>

```haskell
class Packet;  
    rand integer src, dst;  
endclass

int r;  
Packet packet_a=new;  
packet_a.rand_mode(0);  
packet_a.src.rand_mode(1);  
r=packet_a.dst.rand_mode();
```
Controlling Constraints

- Controlling constraints with `constraint_mode()`
  - Can be used to control whether a constraint is active or inactive
    - When a constraint is inactive is not considered by `randomize()`
    - All constraints are initially active
    - `constraint_mode` method is built in and cannot be overridden

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OFF</td>
<td>Sets the specified constraint block to inactive so that it is not enforced by subsequent calls to <code>randomize()</code> method</td>
</tr>
<tr>
<td>1</td>
<td>ON</td>
<td>Sets the specified constraint block to active so that it is considered by subsequent calls to <code>randomize()</code> method</td>
</tr>
</tbody>
</table>

```pascal
class Packet;
    rand integer src, dst;
    constraint filter{src>2*dst;}
endclass

function integer toggle_rand (Packet p);
    if (p.filter.constraint_mode()==1)
        p.filter.constraint_mode(0);
    else
        p.filter.constraint_mode(1);
    toggle_rand=p.randomize();
endfunction
```
Thank You
Constraints

- Iterative constraints

```c
int A [2][3][4];
```

```c
foreach (A[i,j,k])...

i iterates from 0 to 1
j iterates from 0 to 2
k iterates from 0 to 3
```

```c
```

```c
foreach (B[q,r,,s])...

q iterates from 5 to 1
r iterates from 0 to 3
s iterates from 2 to 1
```
Constraints

• Functions in constraints
  ◆ Some properties are unwieldy or impossible to express in a single expression
    ▶ For instance computing the sum of one’s in a packed array uses a loop
    ▶ Without the loop the loop will have to be unrolled
  ◆ SystemVerilog allows constraint expressions to include function calls
    ▶ Functions cannot contain output or ref arguments
    ▶ Functions should be automatic
    ▶ Functions that appear in constraints cannot modify the constraints
    ▶ Functions shall be called before constraints are solved, and their return values shall be treated as state variables
    ▶ Random variables used as function arguments shall establish an implicit variable ordering or priority

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

Forces y to be solved before x
Constraints

• Constraint guards
  ❍ Constraint guards are predicate expressions that function as guards against creation of constraints
    ▶ They are not logical relations that have to be satisfied by the constraint solver
    ▶ Prevents the solver from generating evaluation errors
  ❍ Constraint guards are solved before the constraints and involve
    ▶ Constants
    ▶ State variables
    ▶ Object handle comparisons

The constraint will fail on the last element due to a non existent handle in the linked list

```cpp
class SList;
    rand int n;
    constraint sort{n<next.n;}
endclass
```

if statement acts as a constraint guard

```cpp
class SList;
    rand int n;
    constraint sort{if(next!=null)n<next.n;}
endclass
```
Constraints

• Constraint guards
  
  Guard expressions can themselves include sub-expressions that result in evaluation errors
  
  A 4-state representation is used where
  
  - 0 : false -> Subexpression evaluates to FALSE
  - 1 : true -> subexpression evaluates to TRUE
  - E: Error -> Subexpression causes an evaluation error
  - R: Random -> Expression includes random variables and cannot be evaluated

```verbatim
class D
  int x;
endclass

class C;
  rand int x,y;
  D a,b;
  constraint C1{(x<y || a.x > b.x || a.x==5) -> x+y==10;}
endclass
```

<table>
<thead>
<tr>
<th>Case</th>
<th>a</th>
<th>b</th>
<th>a.x</th>
<th>b.x</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!0</td>
<td>0</td>
<td>5</td>
<td>error</td>
<td>x+y=10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
<td>error</td>
<td>-</td>
<td>always error</td>
</tr>
<tr>
<td>3</td>
<td>!0</td>
<td>!0</td>
<td>10</td>
<td>20</td>
<td>(x&lt;y) -&gt; (x+y==10)</td>
</tr>
</tbody>
</table>
Constraints

• Constraint guards

```plaintext
class D
    int x;
endclass

class C;
    rand int x,y;
    D a,b;
    constraint C1{(x<y && a.x > b.x && a.x==5) -> x+y==10;}
endclass
```

<table>
<thead>
<tr>
<th>Case</th>
<th>a</th>
<th>b</th>
<th>a.x</th>
<th>b.x</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!0</td>
<td>0</td>
<td>5</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
<td>error</td>
<td>-</td>
<td>always error</td>
</tr>
<tr>
<td>3</td>
<td>!0</td>
<td>!0</td>
<td>5</td>
<td>2</td>
<td>( x&lt;y ) -&gt; (x+y==10)</td>
</tr>
</tbody>
</table>