INF5430

SystemVerilog for Verification

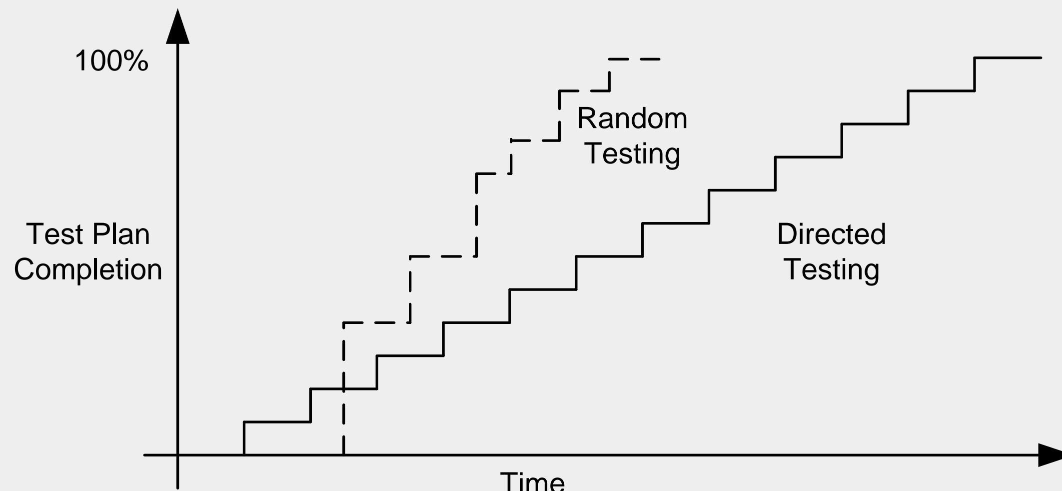
Chapter 6.1-12

Randomization

Chapter 6: Randomization

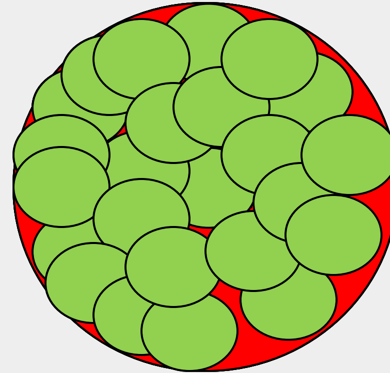
- Directed testing:
  - Checks only anticipated bugs
  - Scales poorly as requirements change
  - Little upfront work
  - Linear progress

- Random testing:
  - Checks unanticipated bugs
  - Scales well as requirements change
  - More upfront work
  - Better than linear progress



6.1 Introduction

- A testbench based on randomization is a shotgun
- The features you are trying to test is the target



- How to cover untested areas?
    - More random testing with tighter constraints
    - Directed testing
- When is testing done?
    - Functional coverage
    - Code coverage

Shotgun Verification or The Homer Simpson Guide to Verification, Peet James

6.2 What to randomize?

Much more than data

1. Device configuration
2. Environment configuration
3. Primary input data
4. Encapsulated input data
5. Protocol exceptions
6. Errors and violations
7. Delays
8. Test order
9. Seed for the random test

6.3 Randomization in SystemVerilog

- Specified within a class along with constraints
- Variable declared with `rand` keyword distributes values uniformly

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

3, 2, 0, 0, 3, 1, ...

- Variable declared with `randc` keyword distributes values cyclically
  - No repeats within an iteration

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

initial permutation 0,3,2,1
→ next permutation 0,3,2,1
→ next permutation 2,0,1,3

- Constraints specified with `constraint` keyword

```
constraint y_c {y >=1;y<3;}
```

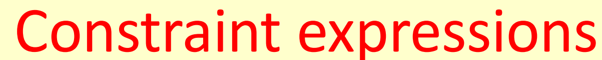
- New values selected when `randomize()` function called
  - Returns 1 if constraints can be solved, 0 otherwise

```
<handle>.randomize();
```

6.3.1 Simple class with Random Variables

```
class Packet;
   rand bit [31:0] src, dst, data[8];
   randc bit [7:0] kind;
   constraint c {src > 10; src < 15;}
endclass

Packet p;
initial begin
   p=new();
   if (!p.randomize())
      $finish;
   transmit(p);
end
```



6.3.2 Checking the result from randomization

- Always check the result of a call to `randomize()`
- Text uses a macro to check the results from randomization

```
`define SV_RAND_CHECK(r) \
   do begin \
     if (!(r)) begin \
       $display("%s:%0d: Randomization failed \"%s\"", \
                `__FILE__, `__LINE__, `"r`"); \
       $finish; \
     end \
   end while (0)
```

```
`SV_RAND_CHECK(p.randomize());
```

```
# test.sv:13: Randomization failed "p.randomize()"
```

6.3.3 The constraint solver

- Solves constraint expressions
- Same seed results in the same random values
- Use different seeds in each nightly regression run.
- Constraints may take a long time to solve
- Solver is specific to each simulator vendor/release.

6.3.4 What can be randomized?

- 2-state variables
- 4-state variables except **no X's or Z's will be created**.
- Integers
- Bit vectors
- Arrays
- Time
- ~~Real, string~~
- ~~Handle in constraint~~

6.4.2 Simple Expressions

- Each constraint expression should contain only 1 relational operator
  - <, <=, ==, >,=>

```
class Order_bad;
   rand bit [7:0] lo, med, hi;
   constraint bad {lo < med < hi;}
endclass
```

```
lo=20,  med=224, hi=164
lo=114, med=39,  hi=189
lo=186, med=148, hi=161
lo=214, med=223, hi=201
```

- Constraint bad is broken down into multiple binary relational expressions from left to right. `lo` and `med` are randomized.
- `lo < med` is evaluated, but not constrained. Results in 0 or 1.
- `hi > 0` or `1` constraint is then evaluated.
- Not what you want!

- Correct constraint:

```
constraint good{lo < med;
                med < hi;}
```

6.4.3 Equivalence Expressions

- Suppose you want to constrain a value to be equal to an expression

```
class order;
   rand bit [7:0] addr_mode, size, len;
   constraint order_c {len == addr_mode*4 + size;}
 endclass
```

- `len` must be declared as random
- Using `=` is a syntax error

6.4.4 Weighted Distributions

- Weighted distributions cause a non-uniform distribution
- Weights do not have to add up to 100% and can be variables
- Cannot be used with randc
- What would this be used for?
    - For a CPU want less or more of a particular opcode
    - For a datapath want max neg, 0, and max pos more often

```
constraint <constraint name> {<variable name> dist {<distribution>}};
```

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

`:=` operator indicates the weight is the same for all values

`:/` operator indicates the weight is distributed across all values

6.4.4 Weighted Distributions := operator

:= operator indicates the weight is the same for all values

```
constraint src_dist { src dist {0:=40, [1:3]:=60} ;}
```

src = 0, weight = 40/220 =18%
src = 1, weight = 60/220 =27%
src = 2, weight = 60/220 =27%
src = 3, weight = 60/220 =27%

6.4.4 Weighted Distributions :/ operator

`:/` operator indicates the weight is distributed across all values

```
constraint dst_dist {dst dist {0:/40,  [1:3]:/60} ;}
```

dst = 0, weight = 40/100 =40%
dst = 1, weight = 20/100 =20%
dst = 2, weight = 20/100 =20%
dst = 3, weight = 20/100 =20%

6.4.4 Weighted Distributions (cont.)

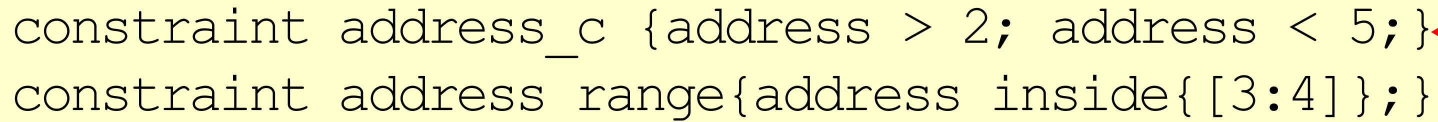
- Weights can be constants, ranges, or variables.
- **Using variables allows the weights to be adjusted on the fly**

```
class BusOp;
   typedef enum {BYTE, WORD, LWRD} length_e;
   rand length_e len;
   bit [31:0] w_byte=1, w_word=3, w_lwrd=5;

   constraint c_len {
       len dist {BYTE := w_byte,
                 WORD := w_word,
                 LWRD := w_lwrd};
   }
endclass
```

6.4.5 Set membership and the `inside` operator

- Alternative to `{var>value1 ; var<value2}` is the `inside` keyword



```
constraint address_c {address > 2; address < 5;}
constraint address_range{address inside{[3:4]};}
```

equivalent

- Using the ! operator can exclude ranges

```
constraint c_range {
    !(c inside{[lo:hi]});
}
```

6.4.6 Using an array in a set

- Suppose you want to create multiple equivalence constraints
- For example: f can only equal 1, 2, 3, 5, 8

```
rand bit [7:0] f;
constraint c_fibonacci {
(f==1) || (f==2) || (f==3) || (f==5) || (f==8));}
```

- Alternate solution is to store the values in an array

```
rand bit [7:0] f;
bit [31:0] vals[]= '{1,2,3,5,8};
constraint c_fibonacci {f inside vals;}
```

- Can specify that values in the array are NOT to be chosen

```
rand bit [7:0] notf;
bit [31:0] vals[]= '{1,2,3,5,8};
constraint c_fibonacci {!(notf inside vals);}
```

6.4.7 Bidirectional Constraints

- Constraint blocks are not procedural but declarative.
- All constraints are active at the same time.

```
rand bit [15:0] r,s,t;
constraint c_bidir {
            r < t;
             s == r;
             t < 10;
             s > 5;}
```

Solution	r	s	t
A	6	6	7
B	6	6	8
C	6	6	9
D	7	7	8
E	7	7	9
F	8	8	9

6.4.8 Implication Constraints

Suppose you want to impose different constraints depending on a var

Solution 1:

```
constraint mode_c {
if (mode == small)
   len < 10;
else if (mode == large)
  len > 100;
}
```

Solution 2:

```
constraint mode_c {
 (mode == small) -> len < 10;
 (mode == large) -> len > 100;
}
```

```
{(a==1)->(b==0)};
```

↓ equivalent

```
{if (a==1) b==0;}
```

↓ equivalent

```
{!(a==1) || (b==0);}
```

↓ equivalent

```
{(a==0) || (b==0);}
```

Solution	a	b
A	0	0
B	0	1
C	1	0

6.4.9 Equivalence operator

- The equivalence operator <-> is bidirectional.
- A<->B is defined as ((A->B) && (B->A))

```
rand bit d, e;
constraint c { d==1 <-> e==1; }
```

6.5 Solution Probabilities

It's important to understand how constraints affect the probability of the solution.

Unconstrained:

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

Solution	x	y	Probability
A	0	0	1/8
B	0	1	1/8
C	0	2	1/8
D	0	3	1/8
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

6.5.2 Implication

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

Solution	x	y	Probability
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

6.5.3 Implication and bidirectional constraints

```
class Imp2;
   rand bit x;
   rand bit [1:0] y;
   constraint c_xy {
      y>0;
      (x==0)->y==0;
   }
endclass
```

Solution	x	y	Probability
A	0	0	0
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	0
F	1	1	1/3
G	1	2	1/3
H	1	3	1/3

6.5.4 Guiding Distribution with solve/before

- `Solve before` tells the solver to solve for 1 variable before another.
- The possible solutions does not change, just the probability.

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

Solution	x	y	Probability
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

solve y before x;

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

Solution	x	y	Probability
A	0	0	1/8
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/4
G	1	2	1/4
H	1	3	1/4

6.6 Controlling Multiple Constraint Blocks

Use the *`constraint_mode()`* function to turn constraints on/off

```
<handle>.constraint_mode(<0/1>);
<handle>.<constraint>.constraint_mode(<0/1>);
```

```
class Packet
    rand bit [31:0] length;
    constraint c_short {length inside {[1:32]};}
    constraint c_long {length inside {[1000:1023]};}
endclass

Packet p:
initial begin
    p=new();
    p.c_short.constraint_mode(0);
    `SV_RAND_CHECK(p.randomize());
    transmit(p);
    ....
    ...
    p.constraint_mode(0);
    p.c_short.constraint_mode(1);
    `SV_RAND_CHECK(p.randomize());
    transmit(p);
end // initial
```

6.7 Valid Constraints

- A suggested technique to creating valid stimulus is to create *valid constraints*
- **Turn the constraint off to test the system's response to invalid stimulus.**
- For example, suppose a read-modify-write command is only valid if the length is a long word.

```
class Transaction;
   typedef enum {BYTE, WORD, LWRD, QWRD} length_e;
   typedef enum {READ, WRITE, RMW, INTR} access_e;
   rand length_e length;
   rand access_e access;

   constraint valid_RMW_LWRD {
   (access == RMW) -> (length == LWRD);
   }
endclass
```

6.8 In-line Constraints

- In-line constraints create constraints outside of the class.
- **Add to existing constraints if they are enabled.**
- For example, a single test needs to be written with tighter than usual address constraints

```
class Transaction;
    rand bit [31:0] addr, data;
    constraint c1 {addr inside{[0:100], [1000:2000]};}
endclass
intitial begin
   Transaction t;
   t=new();
   `SV_RAND_CHECK(t.randomize() with {addr >=50; addr <=1500;
                                       data <10;});
   driveBus(t);
   `SV_RAND_CHECK(t.randomize() with {addr ==2000; data >10;});
   driveBus(t);
end
```

6.9 pre_randomize/post_randomize

- Implicitly called before/after every call to `randomize()`
- `void` function
  - Cannot consume time.
  - Can only call other functions.
  - Does not return a value
- Overload to add your functionality
- `post_randomize()` is good for cleaning up

6.11 Constraint Tips and Techniques

- Instead of hardcoding constraints use variables with defaults
  - Allows the constraint to be modified without modifying the class
  - Allows invalid stimulus to be generated

```
class Packet;
  rand bit [31:0] length;
  constraint c_length {
    length inside {[1:100]};
  }
endclass
```

```
class Packet;
  rand bit [31:0] length;
  int max_length= 100;
  constraint c_length {
  length inside {[1:max_length]};
  }
endclass
```

```
initial begin
   Packet p1 = new();
   p1.max_length = 200;
   p1.randomize();
end
```

6.11.2 Using Nonrandom Values

- `constraint_mode()` turns on/off constraints
- `rand_mode()` makes a variable or every variable in an object non-random

```
class Packet;
   rand bit [7:0] length;
   constraint c_length{length > 0;}
   ..... // Other constraints depending on length
endclass

initial begin
   Packet p = new();
   `SV_RAND_CHECK(p.randomize());
   p.length.rand_mode(0);
   p.length = 42; (or: 0;)
   `SV_RAND_CHECK(p.randomize());
   p.rand_mode(0);
end
```

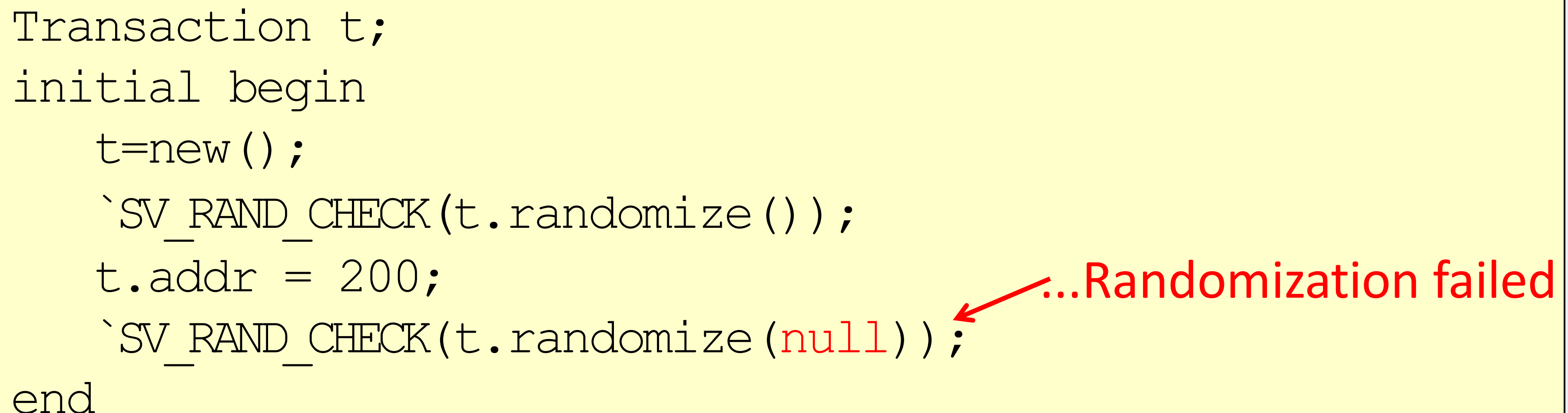
Make length nonrandom

Create an invalid length zero

Value for length will be included in constraint solution

6.11.3 Checking Values using Constraints

- If you change the value of random variables how do you know all your random variables are still valid?
- Use a call to `<handle>.randomize(null)` to check.



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

Transaction t;
initial begin
   t=new();
   `SV_RAND_CHECK(t.randomize());
   t.addr = 200;
   `SV_RAND_CHECK(t.randomize(null));
end
```

6.11.4 Randomizing Individual Variables

Can pass variables to `randomize()` to randomize only a subset of variables

```
class Rising;
   bit [7:0] low;
   rand bit [7:0] med, hi;
   constraint up { low < med; med < hi; }
endclass

initial begin
   Rising r;
   r = new();
   `SV_RAND_CHECK(r.randomize());
   `SV_RAND_CHECK(r.randomize(med));
   `SV_RAND_CHECK(r.randomize(low)); // Surprisingly!!
   `SV_RAND_CHECK(r.randomize(low, med));
end
```

6.11.5 Turn Constraints Off and On

- Use many simple constraints instead of 1 complex constraint
- Turn on the constraints needed

```
class Instruction;
   typedef enum {NOP, HALT, CLR, NOT} opcode_e;
   rand opcode_e opcode;
   bit [1:0] n_operands;
   constraint c_operands{
      if (n_operands == 0)
            (opcode == NOP) || (opcode == HALT);
      else if (n_operands == 1)
            (opcode == CLR) || (opcode == NOT);
   .....
   }
endclass
```

6.11.5 Turn Constraints Off and On (cont.)

```
class Instruction;
typedef enum {NOP, HALT, CLR, NOT} opcode_e;
   rand opcode_e opcode;
   constraint c_no_operands{
      (opcode == NOP) || (opcode == HALT);}
   constraint c_one_operand{
      (opcode == CLR) || (opcode == NOT);}
   .....
   }
endclass

initial begin
   Instruction instr = new();
   instr.constraint_mode(0);
   instr.c_no_operands.constraint_mode(1);
   `SV_RAND_CHECK(instr.randomize());
end
```

6.12 Common Randomization Problems

- Using a signed variable isn't an issue if you control the values

```
for (int i=0;i<=5;i++)
```

- However, a randomized signed variable will produce negative values

```
class SignedVars;
   rand byte pkt1_len, pk2_len;
   constraint total_len {pkt1_len + pk2_len == 64;}
endclass
```

- Some valid solutions of `{pkt1_len, pkt2_len}` are:

```
(32,32)
(2,62)
(-63, 127)
```

6.12.1 Use Signed Values with care

- Might be temped to declare `pkt1_len, pk2_len` as large unsigned

```
class Vars32;
   rand bit [31:0] pkt1_len, pk2_len;
   constraint total_len {pkt1_len + pk2_len == 64;}
endclass
```

- A valid solution of `{pkt1_len, pkt2_len}` is

```
(32'h80000040, 32'h80000000) = 32'h40=32'd64
```

- One solution is to constrain the max values of `pkt1_len` and `pk2_len`
- Best solution is to only use values as wide as required

```
class Vars8;
   rand bit [7:0] pkt1_len, pkt2_len;
   constraint total_len {pkt1_len + pkt2_len == 9'd64;}
endclass
```

Constraint Exercise 1

Write the SystemVerilog code for the following items:
1) Create a class `Exercise1` containing two variables, 8-bit `data` and 4-bit `address`. Create a constraint block that keeps `address` to 3 or 4.
2) In an `initial` block, construct an `Exercise1` object and randomize it. Check the status from randomization.

Constraint Exercise 1 solution

Write the SystemVerilog code for the following items:
1) Create a class `Exercise1` containing two variables, 8-bit `data` and 4-bit `address`. Create a constraint block that keeps `address` to 3 or 4.
2) In an `initial` block, construct an `Exercise1` object and randomize it. Check the status from randomization.

```
class Exercise1;
    rand bit [7:0] data;
    rand bit [3:0] address;
    constraint address_c {
       address > 2;
       address < 5;
    }
endclass

initial begin
  Exercise1 MyExercise1;
  MyExercise1 = new;
  `SV_RAND_CHECK(MyExercise1.randomize());
end
```

```
// or
//   ((address==3) || (address==4));
// or
//   address inside {[3:4]};
```

Constraint Exercise 2

Modify the solution for `Exercise1` to create a new class `Exercise2` so that:

1. `data` is always equal to 5
2. Probability of `address = 4’d0` is 10%
3. Probability of `address` being between [1:14] is 80%
4. Probability of `address = 4’d15` is 10%

Demonstrate its usage by generating 20 new `data` and `address` values and check for error.

Constraint Exercise 2 solution

Modify the solution for `Exercise1` to create a new class `Exercise2` so that:

1. `data` is always equal to 5
2. Probability of `address = 4'd0` is 10%
3. Probability of `address` being between [1:14] is 80%
4. Probability of `address = 4'd15` is 10%

```
package my_package;

  class Exercise2;
    rand bit [7:0] data;
    rand bit [3:0] address;
    constraint data_c {data == 5;}
    constraint address_dist {
      address dist {0:=10,
                    [1:14]:/80,
                    15:=10};
    }

    function void print_all;
      $display("data = %d, address = %d", data, address);
    endfunction

  endclass

endpackage
```

The := operator when the weight is the same for every specified value in the range.

The :/ operator when the weight is to be equaly divided between all values.

Constraint Exercise 2 solution cont.

Demonstrate its usage by generating 20 new `data` and `address` values and check for error.

```
program automatic test;

   import  my_package::*;

   initial begin
      Exercise2 MyExercise2;
      repeat (20) begin
         MyExercise2 = new;
         `SV_RAND_CHECK(MyExercise2.randomize());
         MyExercise2.print_all();
      end
   end

endprogram
```

Constraint Exercise 3

```
class Stim;
   const bit [31:0] CONGEST_ADDR = 42;
   typedef  enum {READ, WRITE, CONTROL} stim_e;
   randc stim_e kind;
   rand bit [31:0] len, src, dst;
   bit congestion_test;

   constraint c_stim {
      len < 1000;
      len > 0;
      if (congestion_test) {
         dst inside {[CONGEST_ADDR-10:CONGEST_ADDR+10]};
         src == CONGEST_ADDR;
      } else
        src inside {0, [2:10], [100:107]};
   }
endclass
```

What are the constraints on len, dst, and src for this code?

Constraint Exercise 3 solution

```
class Stim;
     const bit [31:0] CONGEST_ADDR = 42;
     typedef  enum {READ, WRITE, CONTROL} stim_e;
     randc stim_e kind;
     rand bit [31:0] len, src, dst;
     bit congestion_test;

     constraint c_stim {
          len < 1000;
          len > 0;
          if (congestion_test) {
               dst inside {[CONGEST_ADDR-10:CONGEST_ADDR+10]};
               src == CONGEST_ADDR;
          } else
            src inside {0, [2:10], [100:107]};
     }
endclass
```

What are the constraints on len, dst, and src for this code?

- len must be between 1 and 999 inclusive
- **if** bit congestion_test is 1 dst must be inside 42-10 =32 to 42+10 (52) and src = 42
- **else** src can take on values 0, 2 to 10, and 100 to 107. dst is unconstrained.

Constraint Exercise 4

For the following class create:

1. A constraint that limits read transactions addresses to the range 0 to 7, inclusive
2. Write behavioral code to turn off the above constraint. Construct and randomize a MemTrans object with an in-line constraint that limits read transaction addresses to the range 0 to 8, inclusive. Test that the in-line constraint is working.

```
class MemTrans;
   rand bit rw; // read if rw = 0, write if rw = 1
   rand bit [7:0] data_in;
   rand bit [3:0] address;
endclass
```

Constraint Exercise 4 solution

A constraint that limits **read transactions** addresses to the range **0 to 7**, inclusive:

```
class MemTrans;
  rand bit rw;   // read if rw = 0, write if rw = 1
  rand bit [7:0] data_in;
  rand bit [3:0] address;
  constraint valid_rw_addr { (rw == 0)->(address inside {[0:7]}); }
endclass // MemTrans
```

or

```
class MemTrans;
  rand bit rw;   // read if rw = 0, write if rw = 1
  rand bit [7:0] data_in;
  rand bit [3:0] address;
  constraint valid_rw_addr { if (!rw) address inside {[0:7]}; }
endclass // MemTrans
```

Constraint Exercise 4 solution cont.

Write behavioral code to turn off the above constraint. Construct and randomize a MemTrans object with an in-line constraint that limits read transaction addresses to the range 0 to 8, inclusive. Test that the in-line constraint is working.

```
MemTrans MyMemTrans;
 initial begin
     MyMemTrans = new();
     MyMemTrans.valid_rw_address.constraint_mode(0);
    `SV_RAND_CHECK(MyMemTrans.randomize() with {(rw == 0)->(address inside {[0:8]});});
   end
```