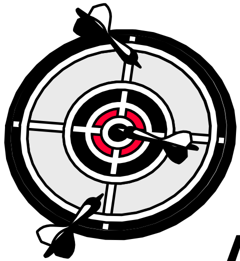
# Agenda

**DAY 2**

| | |
|---|---|
| 5 | Concurrency |
| 6 | Object Oriented Programming (OOP) – Encapsulation |
| 7 | Object Oriented Programming (OOP) – Randomization |

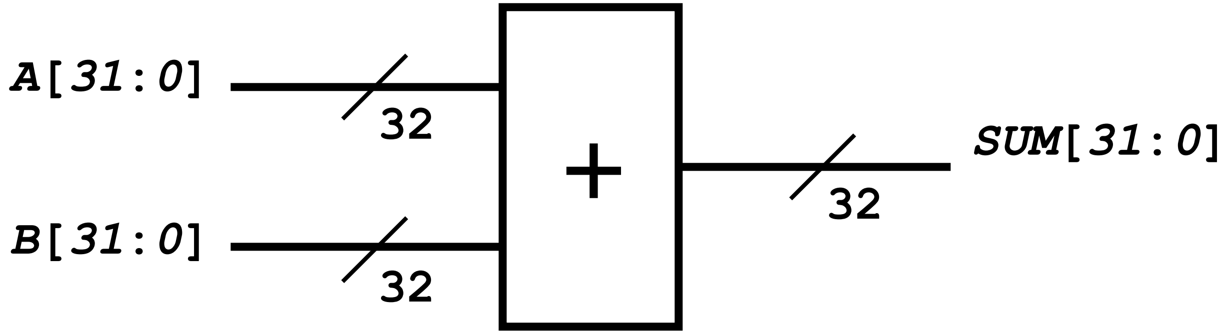
# Unit Objectives



**After completing this unit, you should be able to:**

- **Explain why randomization is needed in verification**
- **Randomize variables**
- **Constrain randomization of variables**

# Alternatives to Exhaustive Testing?

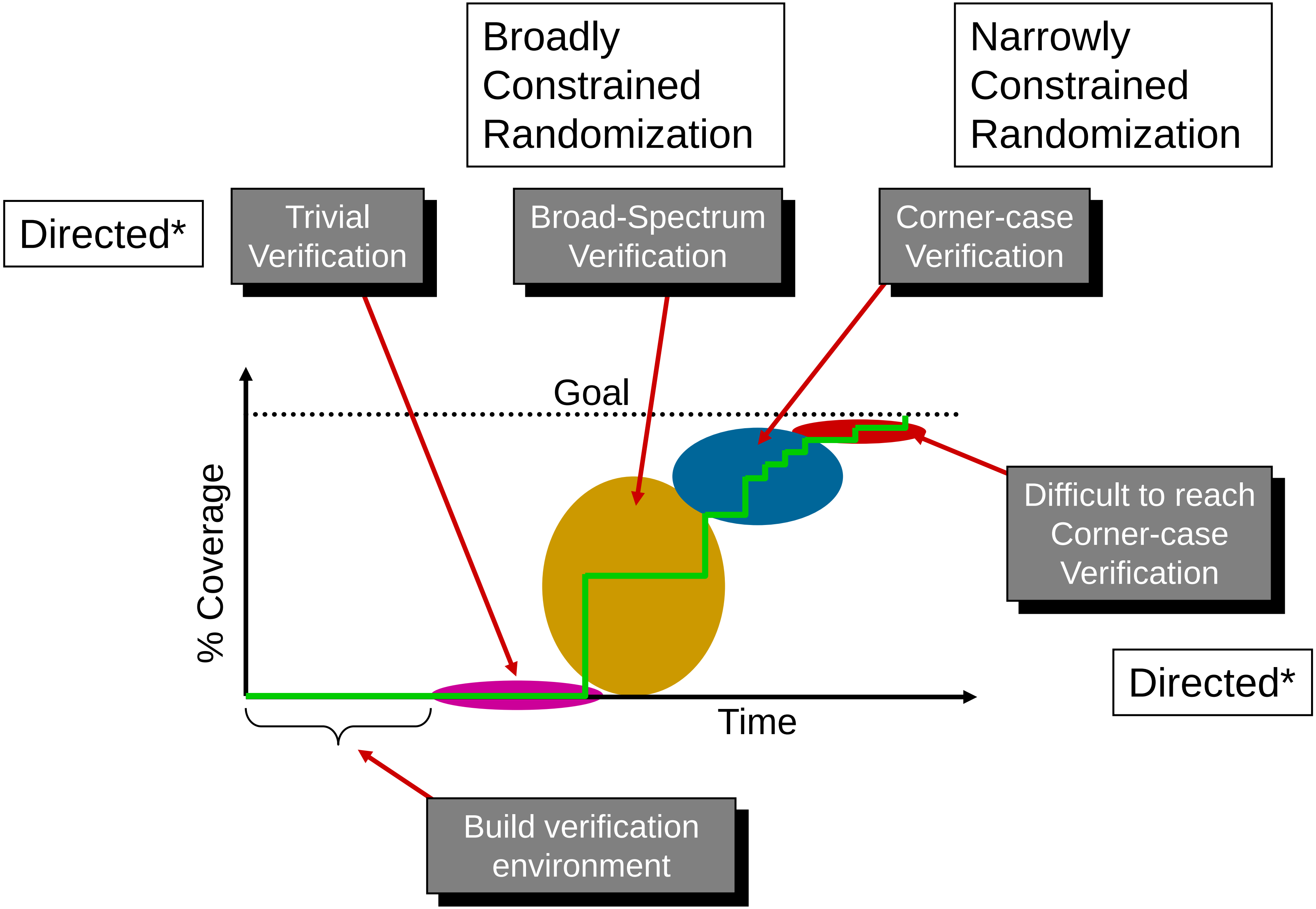



- 32-bit adder example: Assume one set of input and output can be verified every 1ns. How long will exhaustive testing take?
  - A day? – A week? – A year?

- **What if exhaustive testing is unachievable?**
  - **Answer: Verify design with a sufficient set of vectors to gain a level of confidence that product will ship with a tolerable field-failure rate.**

- **Best known mechanism is randomization of data**

# When Do We Apply Randomization?



# OOP Based Randomization

- **In SystemVerilog, randomization is achieved via classes**
  - `randomize()` **function built into every class**
- **Two types of random properties are supported:**
  - `rand` - Values can repeat without exhausting all possible values - Think “rolling dice”
  - `randc` - Exhaust all values before repeating any value -Think “picking a card from a deck of cards”
    - Can be as large as 32-bits in VCS
- **When the class function** `randomize()` **is called:**
  - Randomizes each `rand` and `randc` property value
    - to full range of its data type if no constraints specified

# Randomization Example

```
class Packet;

    randc bit[3:0] sa, da;
    rand  bit[7:0] payload[];

    function Packet copy(...);
       ...;
    endfunction: copy
endclass: Packet
```

1

Declare random properties in class

```
program automatic test;
 int run_for_n_pkts = 100;
 Packet pkt = new();
  initial begin
    ...
    repeat (run_for_n_pkts) begin
       if(!pkt.randomize()) ...;
       fork
           send();
           recv();
       join
       check();
    end
  end
endprogram: test
```

2

Construct an object to be randomized

3

Randomize content of object

# Controlling Random Variables(1/2)

- How do you control the value range for `sa` and `da`?
- How do you control the size of `payload[]`?

```
program automatic test;
  class Packet;
    randc bit[3:0] sa, da;
    rand bit[7:0] payload[];
    function void display();
      $display("sa = %0d, da = %0d", sa, da);
      $display("size of payload array = %0d", payload.size());
      $display("payload[] = %p", payload);
    endfunction: display
  endclass: Packet
  initial begin
    Packet pkt = new();
    if(!pkt.randomize()) $finish;
    pkt.display();
  end
endprogram: test
```

What does `pkt.display()` show for `sa`, `da` and `payload.size()`?

What if `sa`, `da` are `int` type?

`rand int sa, da;`

# Controlling Random Variables(2/2)

- **Randomization controlled by `constraint` block**

```
class Packet;
  rand bit[3:0] sa, da;  rand bit[7:0] payload[];

  constraint corner_test {
    sa == 12;     //equality operator, not assignment
    da inside {2,4,[6:10]};   //set membership
    payload.size() >= 2;      //array aggregate
    payload.size() <= 4;
  }
  constraint valid {…}
endclass: Packet
```

- Constraints support only 2-state values
- Multiple constraint blocks may be defined
- Constraint expression <u>must</u> return true or false

```
constraint single_sa { sa = 12; } // Syntax error
```

# SystemVerilog Constraints

- **Relational Operators**

```
constraint single_sa {
   sa == 12;
   da < sa ;
}
```

- **Set Membership**
  - Select from a list or set with keyword **inside**

```
constraint Limit1 {
   sa inside {[5:7], 10, 15 };
   // 5,6,7,10,15 equally weighted probability
}
```

  - Excluded from a specified set with **!**

```
constraint Limit2 {
   !( sa inside { [1:10], 15 } );
   // 0,11,12,13,14 equally weighted probability
}
```

# Weighted Constraints

- **Constraint values can also be weighted over a specified range using keyword `dist` and:**
  - `:=` (apply the same weight to all values in range)

```
constraint Limit {
  sa dist {[5:7]:=30, 9:=20};
}
// 5,6,7 = weight 30 each
// 9 = weight 20
```

equal weights

  - `:/` (divide the weight among all values in range)

```
constraint Limit {
  da dist {[5:7]:/30, 9:=20};
}
// 5,6,7 = weight 10 each
// 9 = weight 20
```

divided weights

# Array Constraint Support

- Members can be constrained within `foreach` loop
- Aggregates can be used to constrain arrays
  - `size()`, `sum()` and more (see release notes)
- Set membership can be used to reference content

```
class Config;
  rand bit[7:0] addrs[10];
  rand bit drivers_in_use[16];
  rand int num_of_drivers, one_addr;
    constraint limit {
      num_of_drivers inside { [1:16] };
      drivers_in_use.sum() with (int'(item)) == num_of_drivers;
      foreach(addrs[idx])
        (idx > 0) -> addrs[idx] > addrs[idx-1];
      one_addr inside addrs;
    }
endclass: Config
```

See Note

# Implication and Order Constraints

- **Implication operators:**
  - **`->`**
  - **`if ( … ) … [ else … ]`**
  - Caution: does not imply solving order

```
typedef enum { low, mid, high, any } AddrTyp_e;
class MyBus;
    rand bit[7:0] addr;
    rand AddrTyp_e atype;
    constraint addr_range {
      (atype == low ) -> addr inside { [0:15] };
      (atype == mid ) -> addr inside { [16:127] };
      (atype == high) -> addr inside { [128:255] };
//    if (atype == low)  addr inside { [0:15] };
//    else if (atype == mid)  addr inside { [16:127] };
//    else if (atype == high) addr inside { [128:255] };
    }
endclass: MyBus
```

# Equivalence Constraints

- **Equivalence operator:**
  - **`<->`**
    - True bidirectional constraint
    - A <-> B means if A is true B must be true and if B is true A must be true
  - Caution: does not imply solving order

```
typedef enum { low, mid, high, any } AddrTyp_e;
class MyBus;
    rand bit[7:0] addr;
    rand AddrTyp_e atype;
    constraint addr_range {
      (atype == low ) <-> addr inside { [0:15] };
      (atype == mid ) <-> addr inside { [16:127] };
      (atype == high) <-> addr inside { [128:255] };
    }
endclass: MyBus
```

# Uniqueness Constraints

- Constrain each variable in a group to be unique after randomization using `unique`

```
class C;
  rand bit [2:0] a[7];
  rand bit [2:0] b;
  constraint cst1 {
    unique { a[0:2], a[6], b };
  }
endclass

C obj = new;
if (!obj.randomize()) $finish;
$display ("a = ", obj.a);
$display ("b = ", obj.b);
```

Array slices allowed

a = ‘{‘h5, ‘h0, ‘h3, ‘h1, ‘h7, ‘h2, ‘h2};
b = 6

# System functions

- **Bit-vector system functions can be used in constraints (Currently VCS only)**
  - treated as an operator/expression instead of a function
    - **`$countbits`**
    - **`$countones`**
    - **`$onehot`**
    - **`$onehot0`**
    - **`$bits`**

```
rand bit [3:0] vector;
constraint cst {  $countones (vector) == 2; }

//same as
constraint cst {
( vector[0] + vector[1] + vector[2] + vector[3] ) == 2; }
```

**Semantic restrictions on function calls in constraints do NOT apply here**

# User-defined Functions in Constraints

- **User-defined functions can be used to constrain variables**
  - See LRM for rules and limitations on functions and randomization order
  - Can also use C functions using DPI 

```
class D;
   rand bit [6:0] a,b;
   rand bit [7:0] c;
   constraint c0 { c == add(a,b); }

   function bit[7:0] add(input bit[6:0] i1, i2);
       return (i1+i2);
   endfunction
endclass
```

# Constraint Solver Order

- **Dictating solver order:**
  - **`randc`** is solved before **`rand`** properties
  - **`solve-before`** construct sets solving order for same type random properties
    - Can not force **`rand`** to be randomized before **`randc`** properties
  - **`$void(`***rand_property***`)`** solves *rand_property* first

```
class MyBus;
  rand bit flag;
  rand bit[11:0] addr;
  constraint addr_range {
    if ( flag == 0 ) addr == 0;
    else addr inside { [1:1024] };
    solve flag before addr; // guidance only
//  solve addr before flag; // what's the difference?
//  solve flag before addr hard; // force order – VCS only
//    if ( $void(flag) == 0 ) addr == 0; // alternative
  }
endclass: MyBus
```

# Inline Constraints

- **Individual invocations of randomize() can be customized using**

```
obj.randomize( ) with { <new constraints> };
```

```
program automatic test;
  class demo;
    rand int x, y, z;
    constraint Limit1 { x > 0; x <= 5; }
    ...
  endclass: demo
  initial begin
    demo obj_a = new();
    //ADD another constraint. Does NOT override Limit1
    if(!obj_a.randomize() with { x > 3 && x < 10; })…;
    ...
  end
endprogram:  test
```

# Soft Constraints

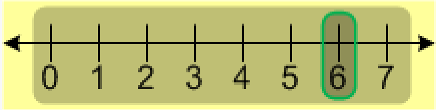
- **Use keyword `soft` when defining soft constraints**
  - Only for **`rand`** variables, not **`randc`** variables

```
class A;
  rand bit [7:0] x;
  constraint A1 { soft x == 6; }
endclass
```

- **Soft constraints are satisfied unless contradicted**
  - By a hard constraint
  - By a soft constraint with higher priority

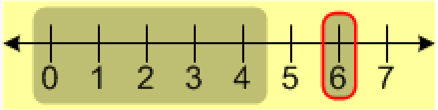
```
A a = new(); //class A defined above
initial begin
  a.randomize() with { x inside {[0:7]}; };
```

x = 6




```
  a.randomize() with { x inside {[0:4]}; };
end
```

$0 \leq x \leq 4$



# Where are Soft Constraints Used?

- **In environment classes**
  - To specify default ranges of random variables
- **In test program**
  - To bias ranges in test

```
class Packet;
  rand bit [11:0] len; rand int min, max;
  constraint len_c { soft len inside {[min:max]};}
  constraint range { soft min == 0; soft max == 10;}
endclass
Packet p,q; int tmin, tmax;
// override class constraints with higher priority constraints
stat = p.randomize()  with {
     soft len inside {[tmin:tmax]};
     }
// or change the object's min and max with hard constraints
stat = q.randomize() with { min==tmin; max==tmax;}
```

# Mutually Constrained Random Variables

- **Constraint limits can be random variables:**

```
class demo;
  rand  bit[7:0] high;
  rand  int  unsigned x;
  constraint Limit
  {
    x > 1;
    x < high;
  }
endclass
```

**What random values are generated for `high`?**

**`randomize()`** will eliminate values {0,1,2} from possible values for high

If there is no legal value for high, then **`randomize()`** prints warning and returns a 0 (properties left unchanged)

Caution: does not imply solving order

# Inconsistent Constraints

```
class demo;
  rand  bit[7:0] high;
  rand  int  unsigned x;
  constraint Limit  {
    x > 1000;
    x <= high;
  }
endclass: demo
```

**What if the constraints cannot be solved?**

`randomize()` produces this simulation error:

```
Solver failed when solving following set of constraints
rand bit[31:0] x; // rand_mode = ON
rand bit[7:0] high; // rand_mode = ON
constraint Limit    // (from this) (constraint_mode = ON) (demo.sv:4)
{
  (x > 1000);
  (x <= high);
}
```

It leaves the object unchanged and returns a status value of 0. Simulation <u>does not</u> stop.

# Effects of Calling `randomize()`

- **When `randomize()` executes, three events occur:**
  - `pre_randomize()` is called
  - Variables randomized
  - `post_randomize()` is called
- `pre_randomize()`
  - **Optional**
  - **Set/Correct constraints**
  - **Example: `rand_mode(0|1)`**
- `post_randomize()`
  - **Optional**
  - **Make corrections after randomization**
  - **Example: CRC**

```
class Packet;
  int test_mode;
  rand bit[3:0]  sa, da;
  rand bit[7:0]  payload[];
       bit[15:0] crc;

  constraint LimitA {
    sa inside { [0:7] };
    da inside { [0:7] };
    payload.size() inside {[2:4]};
  }

  function void pre_randomize();
    if(test_mode) sa.rand_mode(0)
  endfunction: pre_randomize

  function void post_randomize();
    gen_crc(); //user method
  endfunction: post_randomize

endclass: Packet
```

# Controlling Randomization at Runtime

- **Turn randomization for properties on or off with:**

**task/function int *object_name.property*.rand_mode ( 0 | 1 );**

1 - enable randomization (default )

0 - disable randomization

If called as function, returns rand_mode state of property (0 or 1)

```
class Node;
  rand int x, y, z;
  constraint Limit1 {
    x inside {[0:16]};
    y inside {[23:41]};
    z < y; z > x;
  }
endclass:  Node
```

```
program automatic test;
initial begin
  Node obj1 = new();
  obj1.x = 0;
  obj1.x.rand_mode(0);
  if(!obj1.randomize()) ...;
end
endprogram: test
```

**Solver still checks *x* is within its constraints**

# Controlling Constraints at Runtime(1/2)

- **Turn constraint blocks on and off with:**

**task/function int *object_name*.*constraint_block_name*.constraint_mode ( *0* | *1* );**

1 - enable constraint (default )

0 - disable constraint

If called as function, return state of constraint (0 or 1)

```
program automatic test;
  class demo;
    rand int x, y, z;
    constraint no_error { x > 0; x <= 5; }
    static constraint with_error { x > 0; x <= 32; }
  endclass: demo
  initial begin
    demo obj_a = new();
    obj_a.no_error.constraint_mode(0); //test with errors
    if(!obj_a.randomize()) ...;
  end
endprogram: test
```

# Constraint Prototypes

- **Can define constraint prototypes in class**
  - Define the constraint in same scope

demo.sv

```
class demo;
  rand int x, y, z;
  extern constraint valid ; //must define
endclass: demo
//extern constraint must be defined later in same scope as class
constraint demo::valid { x > 0; y >= 0; z % x == 0; }
```

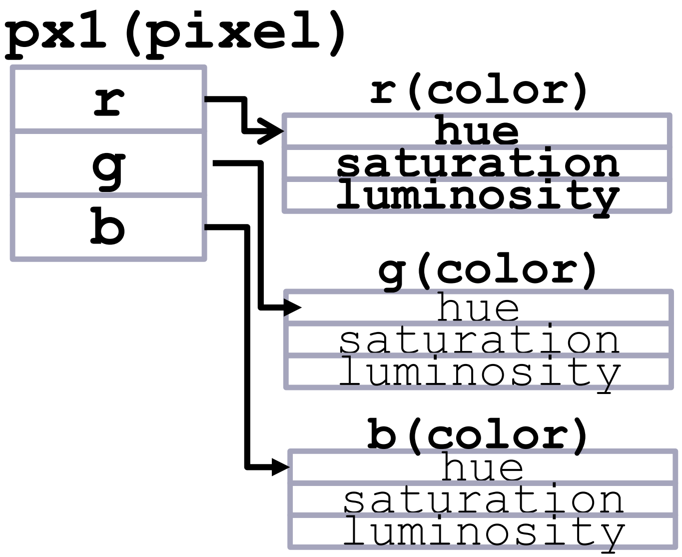
test.sv

```
program automatic test_corner_case;
`include "demo.sv"
initial begin
    demo obj_a = new();
    if(!obj_a.randomize()) ...;
  end
endprogram: test_corner_case
```

# Nested Objects with Random Variables

`randomize( )` follows a linked list of object handles, randomizing each linked object to the end of the list.

```
program automatic test;
  class color
    rand int hue, saturation, luminosity;
  endclass: color
  class pixel;
    rand color r, g, b;
    ...
  endclass: pixel
  initial begin
    pixel px1 = new();
    px1.r = new();
    px1.g = new();
    px1.b = new();
    ...
    if (!px1.randomize())...;
  end
endprogram: test
```




This will randomize objects px1 and px1.r, px1.g, px1.b

# std::randomize()

- **`std::randomize()` for variables outside classes**
  - Very fast performance in VCS
  - **`std::`** is optional
- **Available in modules, functions, tasks, and classes**
  - Randomization using ***`obj.randomize()`*** is still preferred

```
program automatic test;
  bit [11:0] addr;
  bit [5:0] offset;
  function bit genConstrainedAddrOffset();
    return std::randomize(addr, offset) with
      {addr > 1000; addr + offset < 2000;};
  endfunction
endprogram: test
```

Constraints inside **`with`**

# Changing the Random Seed at Simulation

- **VCS allows you to provide an initial seed during simulation with the following options to simulator**
  - `+ntb_random_seed = <seed>`
    - Set the initial seed to <seed>

    ```
    % simv <other_opts> +ntb_random_seed=123
    ```
  - `+ntb_random_seed_automatic`
    - Create a unique seed for each simulation by combining the time of day, hostname and process id

    ```
    % simv <other_opts> +ntb_random_seed_automatic
    ```
  - Seed appears in simulation log and coverage report
- **VCS allows you to query for the initial simulation seed**
  - `$get_initial_random_seed();`
- **VCS allows you to save simulation log messages to a file**

  ```
  % simv <other_opts> -l simv.log
  ```

# Quiz Time

# Randomization: Quiz 1

```
program automatic test1;

class abc;
    rand int a;
    constraint c1 {a inside {[1:4]};}
    constraint c2 {a inside {[5:8]};}
endclass

initial begin

  abc o1 = new();
  o1.randomize();
  $display("test: o1 = %p", o1);

end

endprogram: test1
```

1. **Will this code compile without errors?**
2. **Will it throw any runtime errors?**
3. **What will the program display?**

# Randomization: Quiz 2

```
program automatic test2;

class abc;
   rand int a;
   constraint c1 {a inside {[1:4]};}
endclass

initial begin

  abc o2 = new();
  o2.randomize() with {a inside {[5:8]};};
  $display("test1: o2 = %p", o2);
end

endprogram: test2
```

1. **Will this code compile without errors?**
2. **Will it throw any runtime errors?**
3. **What will the program display?**

# Randomization: Quiz 3

```
program automatic test3;

class abc;
   rand int a;
   constraint c1 {soft a inside {[1:4]};}
endclass

initial begin

  abc o3 = new();
  o3.randomize() with {a inside {[5:8]};};
  $display("test3: o3 = %p", o3);

end

endprogram: test3
```

1. **Will this code compile without errors?**
2. **Will it throw any runtime errors?**
3. **What will the program display?**

# Randomization: Quiz 4

```
program automatic test4;

class abc;
   int a;
   constraint c {a inside {[0:4]};}
endclass: abc

initial begin

  abc o4 = new();
  o4.a = 6 ;
  o4.randomize();

  $display("test4: o4 = %p", o4);

end

endprogram: test4
```

1. Will this code compile without errors?
2. Will it throw any runtime errors?
3. What will the program display?

# Randomization: Quiz 5

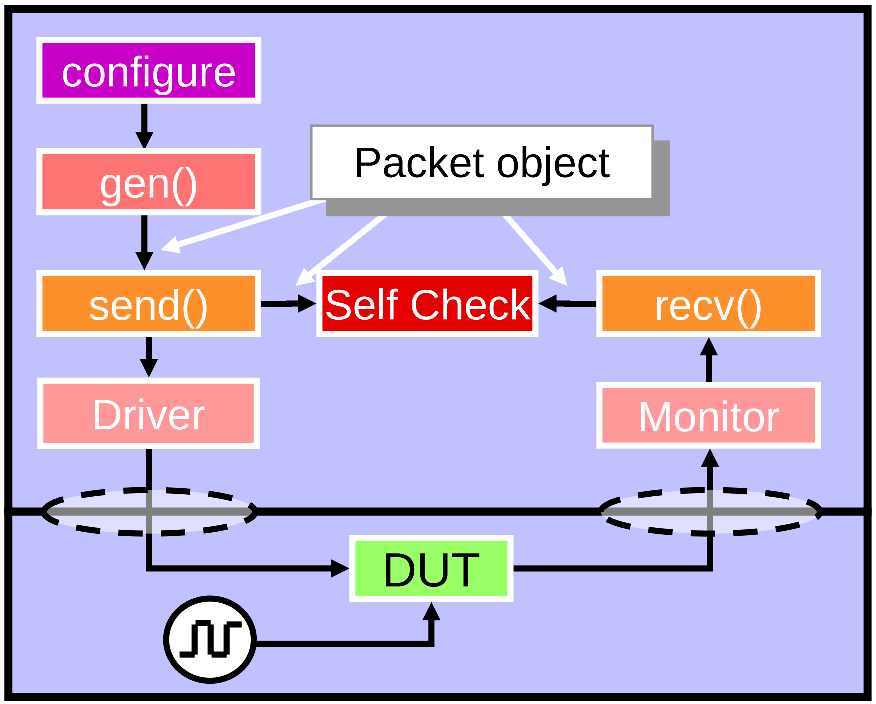
```
program automatic test5;

class abc;
   rand int a;
   constraint c {a inside {[0:4]};}
endclass: abc

initial begin

  abc o5 = new();
  o5.a = 6
  o5.randomize();

  $display("test5: o5 = %p", o5);

end

endprogram: test5
```

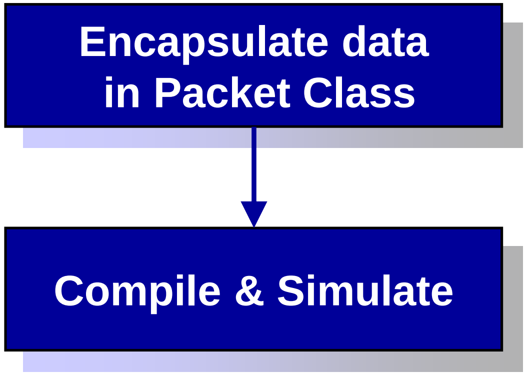
1. Will this code compile without errors?
2. Will it throw any runtime errors?
3. What will the program display?

# Lab 4 Introduction

90 min

## Encapsulate data in Packet Class



# Unit Objectives Review

**Having completed this unit, you should be able to:**

- **Explain why randomization is needed in verification**
- **Randomize variables**
- **Constrain randomization of variables**

# Appendix

`struct` **Ramdomization**

**Soft Constraints: Rules and Management**

**Random Stability**

**Constraint Debug and Profiling**

**Methodology and Best Practices**

## `struct` Ramdomization

# struct Randomization Example (1/2)

- **struct inside class**

```
typedef struct {
  rand int x;
  int y;
} ST0;

typedef struct packed {
 int x;
 int y;
} ST1;

class C;
  rand ST0 st0;
  rand ST1 st1;
  constraint cst0 { st0.x == 10; }
  constraint cst1 { st0.x == st1.x; }
endclass
```

Not rand

x and y randomized because packed

Output:
st0: x:10, y:0
st1: x:10, y:8

```
program automatic test;
  C obj = new;
  initial begin
     obj.randomize;
     $display("%p", obj);
  end
endprogram
```

# `struct` Randomization Example (2/2)

- **Object inside `struct`**

```
class C;
  rand int x;
  constraint cst { x == 123; }
endclass

typedef struct {
  rand int y;
  rand C c0;
} ST1;
```

```
program automatic test;
initial begin
   ST1 s;
   s.c0 = new;
   std::randomize(s) with {
     s.c0.x == s.y;
   };
   $display("%p, %p", s, s.c0);
end
endprogram
```

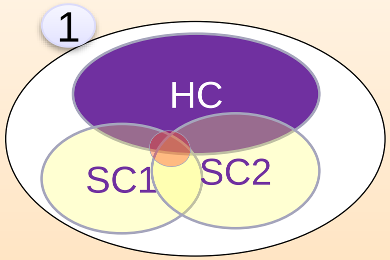
Auto-allocates `s.y`
`c0` must be constructed

Output:
y:123, c0:{ ref to class C}
x:123

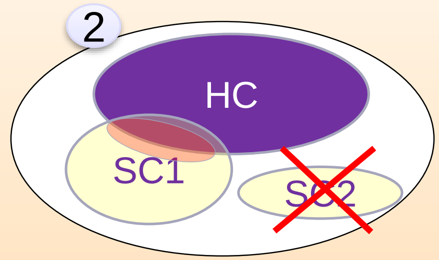
# Soft Constraints: Rules and Management

# How do soft constraints work?

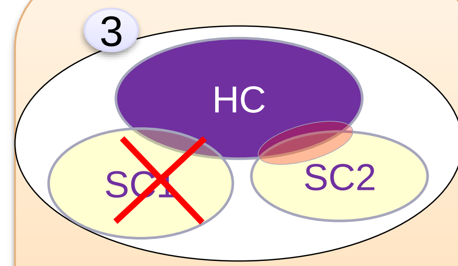
Only ***Contradictions*** can lower solving priorities



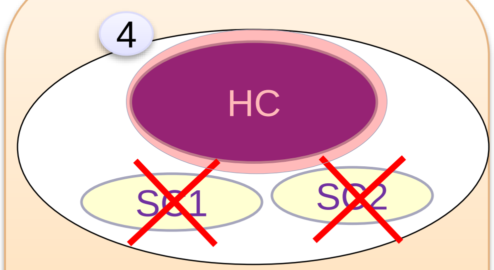

HC:   a inside {[0:100]}
SC1: a inside {[-100:20]}
SC2: a inside {[-20:40]}



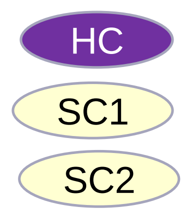

HC:   a inside {[0:100]}
SC1: a inside {[-100:20]}
SC2: a inside {[120:140]}




HC:   a inside {[0:100]}
SC1: a inside {[-100:20]}
SC2: a inside {[60:120]}




HC:   a inside {[0:100]}
SC1: a inside {[-100:-120]}
SC2: a inside {[-80:-60]}




Hard constraint
Soft constraint #1
Soft constraint #2

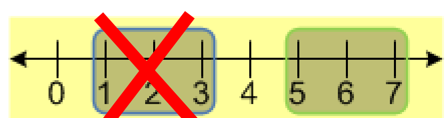
# Priority of Soft Constraints(1/5)

High

1. Inline soft constraints with the randomize call (1)
2. Soft constraints in the randomized class (2)
3. Constraints of embedded member objects
4. Parents of a class in the inheritance graph

Low

```
class A;
    rand bit [2:0] n;
    constraint A1 {
        soft n inside {[1:3]};    2
    }
endclass
program automatic test;
    A a = new();
    initial
        a.randomize() with {
            soft n > 4;           1
        };
endprogram
```




Note: The last declared constraint within the same scope has higher priority

# Priority of Soft Constraints(2/5)

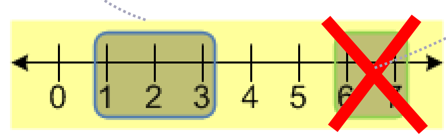
High

- Inline soft constraints with the randomize call
- Soft constraints in the randomized class
- Constraints of embedded member objects
- Parents of a class in the inheritance graph

Low

❶❷

```
class A;
   rand bit [2:0] x;
   constraint A1 {
        ❷soft x > 5;
        ❶soft x inside {[1:3]};
     }
endclass
program automatic test;
   A a = new();
     initial
          a.randomize();
endprogram
```




Note: The last declared constraint within the same scope has higher priority

# Priority of Soft Constraints(3/5)

High

- Inline soft constraints with the randomize call
- Soft constraints in the randomized class 1
- Constraints of embedded member objects 2 3
- Parents of a class in the inheritance graph

Low

```
class A;
3  rand bit [7:0] n;
   constraint A1 { soft n inside {[5:9]};}
endclass
```

```
class B;
   rand bit [7:0] m;
2  constraint B1 { soft m inside {[0:4]};}
endclass
```

```
class C;
   rand A objA = new();
   rand B objB = new();
1  constraint C1 {soft objA.n == objB.m;}
     …
endclass
```

```
program automatic test;
   C c = new();
   initial c.randomize();
endprogram
```

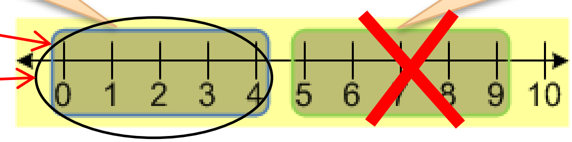
objB.B1

objA.A1

objA.n ∈

objB.m ∈

objA.n == objB.m




Note: The last declared constraint within the same scope has higher priority

# Priority of Soft Constraints(4/5)

High

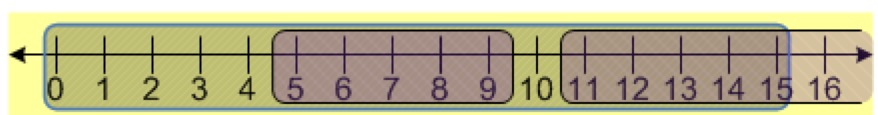
- Inline soft constraints with the randomize call
- Soft constraints in the randomized class
- Constraints of embedded member objects
- Parents of a class in the inheritance graph

Low

```
class A;
    rand bit [7:0] n;
    constraint A1 {
        soft n > 10;
    }
endclass
```

```
class B extends A;
    constraint B1 {
        n < 16;
        soft n inside {[5:9]};
    }
```

```
program automatic test;
    B b = new();
    initial  b.randomize();
endprogram
```




Note: The last declared constraint within the same scope has higher priority

# Priority of Soft Constraints(5/5)

High

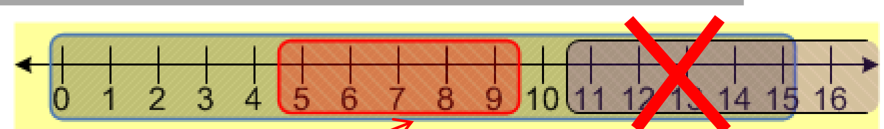
- Inline soft constraints with the randomize call
- Soft constraints in the randomized class ❶
- Constraints of embedded member objects
- Parents of a class in the inheritance graph ❷

Low

```
class A;
      rand bit [7:0] n;
      constraint A1 {
❷             soft n > 10;
      }
endclass
```

```
class B extends A;
   constraint B1 {
        n < 16;
❶       soft n inside {[5:9]};
   }
```

```
program automatic test;
      B b = new();
      initial  b.randomize();
endprogram
```



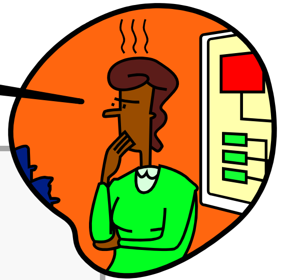

***b.n*** ∈

Note: The last declared constraint within the same scope has higher priority

# Disabling Soft Constraints(1/3)

- **Soft constraints sometimes can narrow down the range of a random variable, even more than needed**

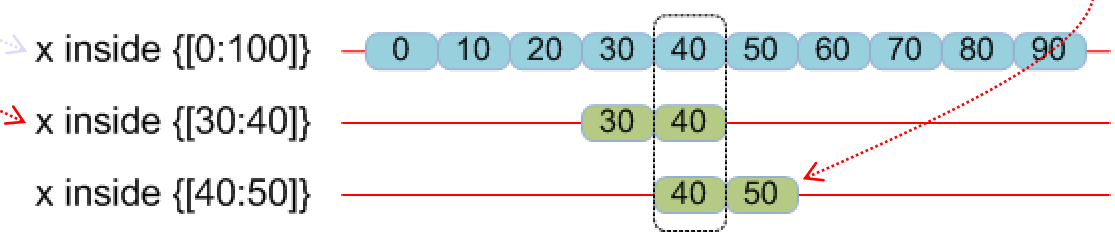
But I do NOT want to keep this soft constraint!!!



```
class my_class;
  rand bit [7:0] x;
  constraint x_constr {
     x inside {[0:100]};
     soft x inside {[30:40]};
   }
endclass: my_class
```

```
my_class obj = new();
initial begin
      obj.randomize with {
      soft x inside {[40:50]};
      };
end
```

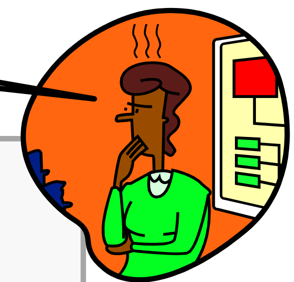
Test



# Disabling Soft Constraints(2/3)

- Use `disable soft` to disable soft constraints

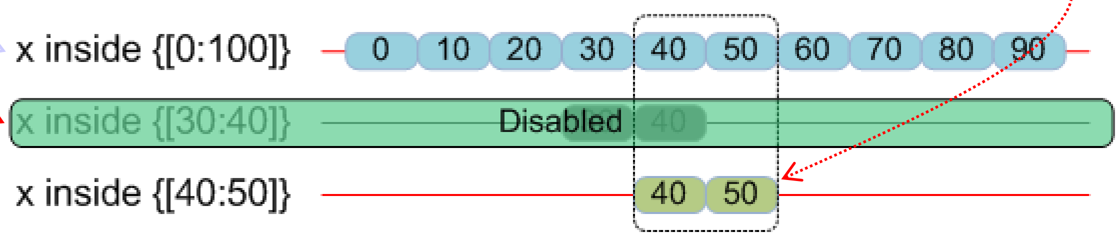
But I do NOT want to keep
this soft constraint!!!



```
class my_class;
  rand bit [7:0] x;
  constraint x_constr {
      x inside {[0:100]};
      soft x inside {[30:40]};
    }
endclass: my_class
```

```
my_class obj = new();
initial begin
    obj.randomize with {
    disable soft x;
    soft x inside {[40:50]};
    };
end
```

Test



# Disabling Soft Constraints(3/3)

```
class sc_class;
   rand bit [7:0] x, y;
   constraint c1 {
      soft x == 10;
   }
   extern constraint c2;
endclass
```

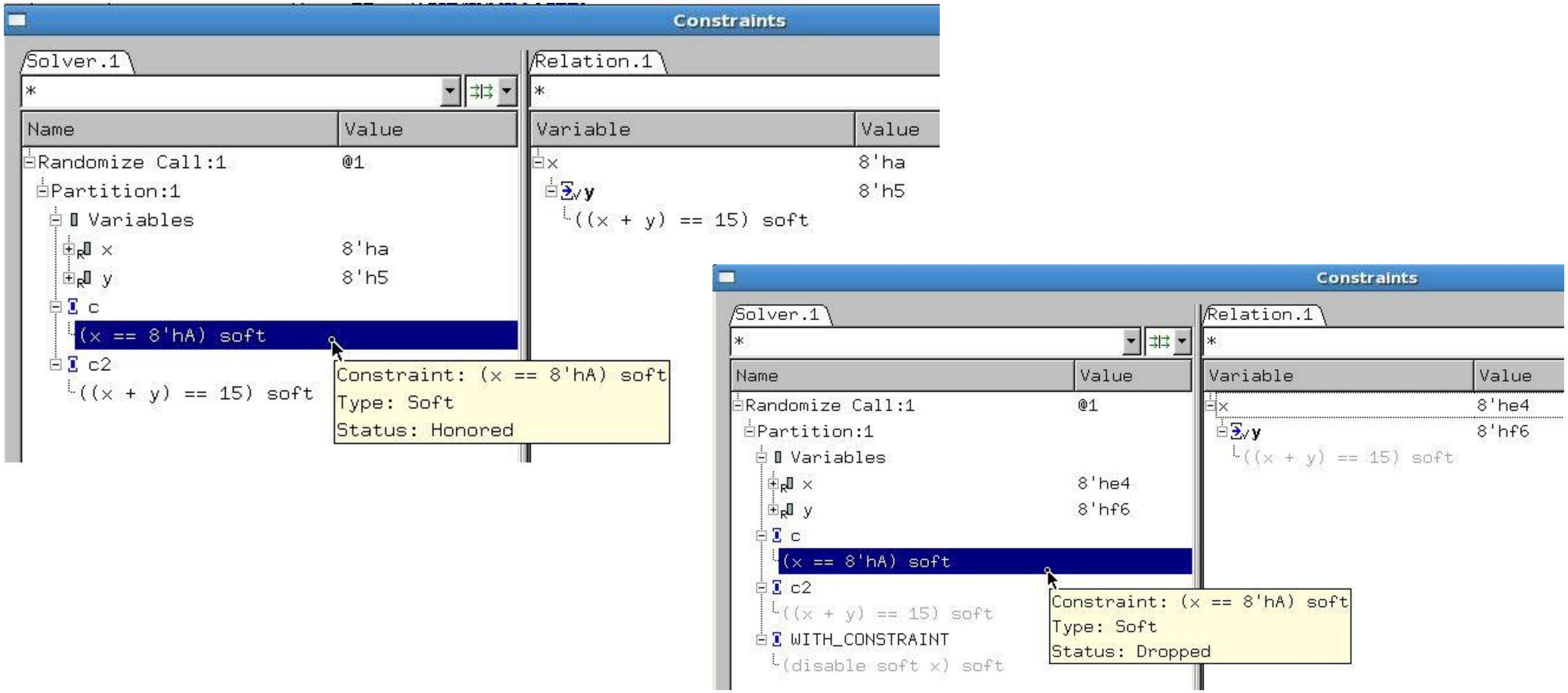
```
program automatic t;
   sc_class o = new();
   initial begin
      o.randomize() with {
        disable soft x;
        };
        …
    end
endprogram
```

```
constraint
sc_class::c2 {
   soft x + y == 15;
}
```

x and y become unconstrained

- A **`disable soft`** constraint on a random variable specifies that all lower priority soft constraints that reference that random variable shall be dropped.

# Debug Soft Constraints using DVE




Constraints debugger allows you to debug soft and hard constraints

- Was a soft constraint dropped or honored?
- Explore relations between the variables
- List all constraints related to some variable
- Find insufficiencies in the constraints
- View initial (all possible values) list of variables

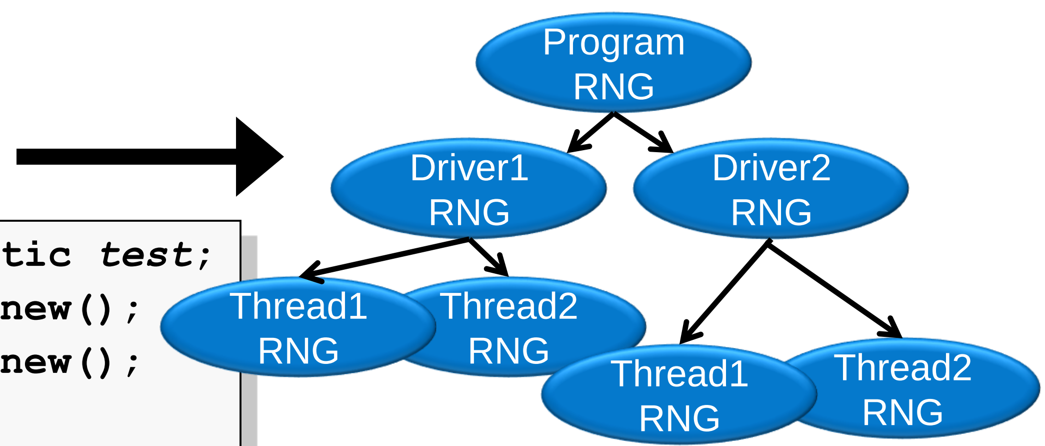
# Random Stability

# Random Stability: Threads and Objects

- **Verilog**
  - Changes create different random values & results
- **SystemVerilog**
  - All modules start with the same seed
    - Modules not recommended for tests – use **`program`**
  - Random Number Generator (RNG) per object, thread
  - RNG are hierarchical seeded from program root
    - Most small code changes give same runtime results
    - Use **`srandom(`*`seed`*`)`** to seed each object or thread if needed

```
task driver::main();
  fork
    thread1();
    thread2();
  join_none
endtask: mai
```

```
program automatic test;
  driver drv1=new();
  driver drv2=new();
  ...
```



# Constraint Debug and Profiling

# Debug - Constraint Conflicts

- **VCS Simplifies Constraint Debug**
  - Incorrect constraints can result in no legal solution
  - Entire problem may be very large, 1000+ constraints
  - VCS Identifies conflicting variables and constraints
  - Problem can be analyzed immediately

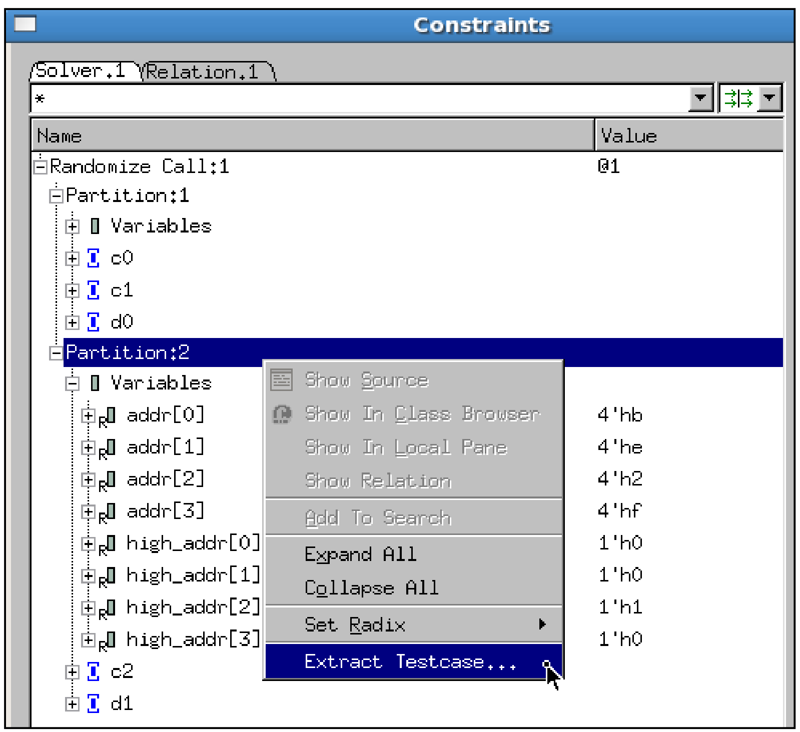
```
Solver failed when solving following set of constraints

rand bit[7:0] a; // rand_mode = ON
rand bit[7:0] b; // rand_mode = ON

constraint cst1  { a > 5;  }              // test.sv:19
constraint cst2  { b > 10; }              // test.sv:20
constraint cst3  { a + b == 12 ; }        // test.sv:21
```

Conflicting Variables Subset

Conflicting Constraints Subset

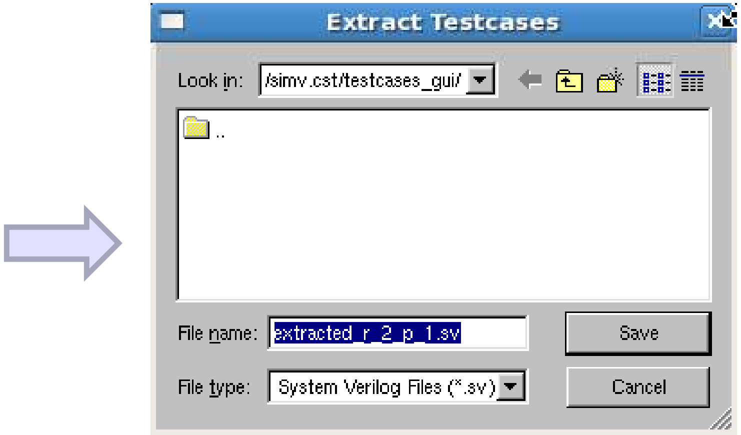
# Interactive Constraint Debug using DVE (1/5)

- **Standalone constraint testcase for a given partition inside the randomize call can be extracted from constraint dialog**



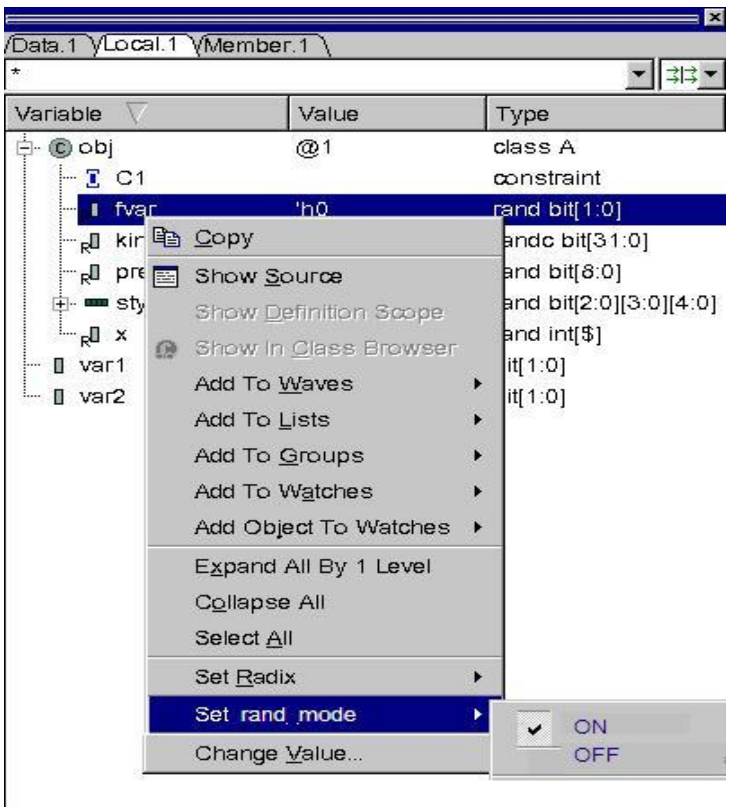

Benefits:

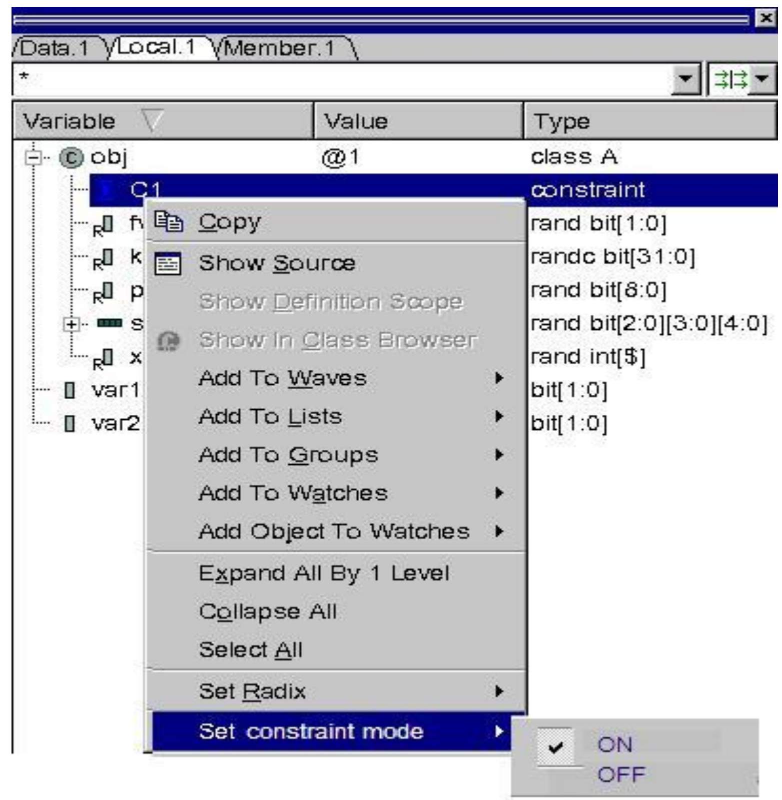
- Standalone testcase for more debug (compile/run outside of original design)
- No need to run simv again



# Interactive Constraint Debug using DVE (2/5)

- **Modification of constraint_mode and rand_mode from within the local pane**

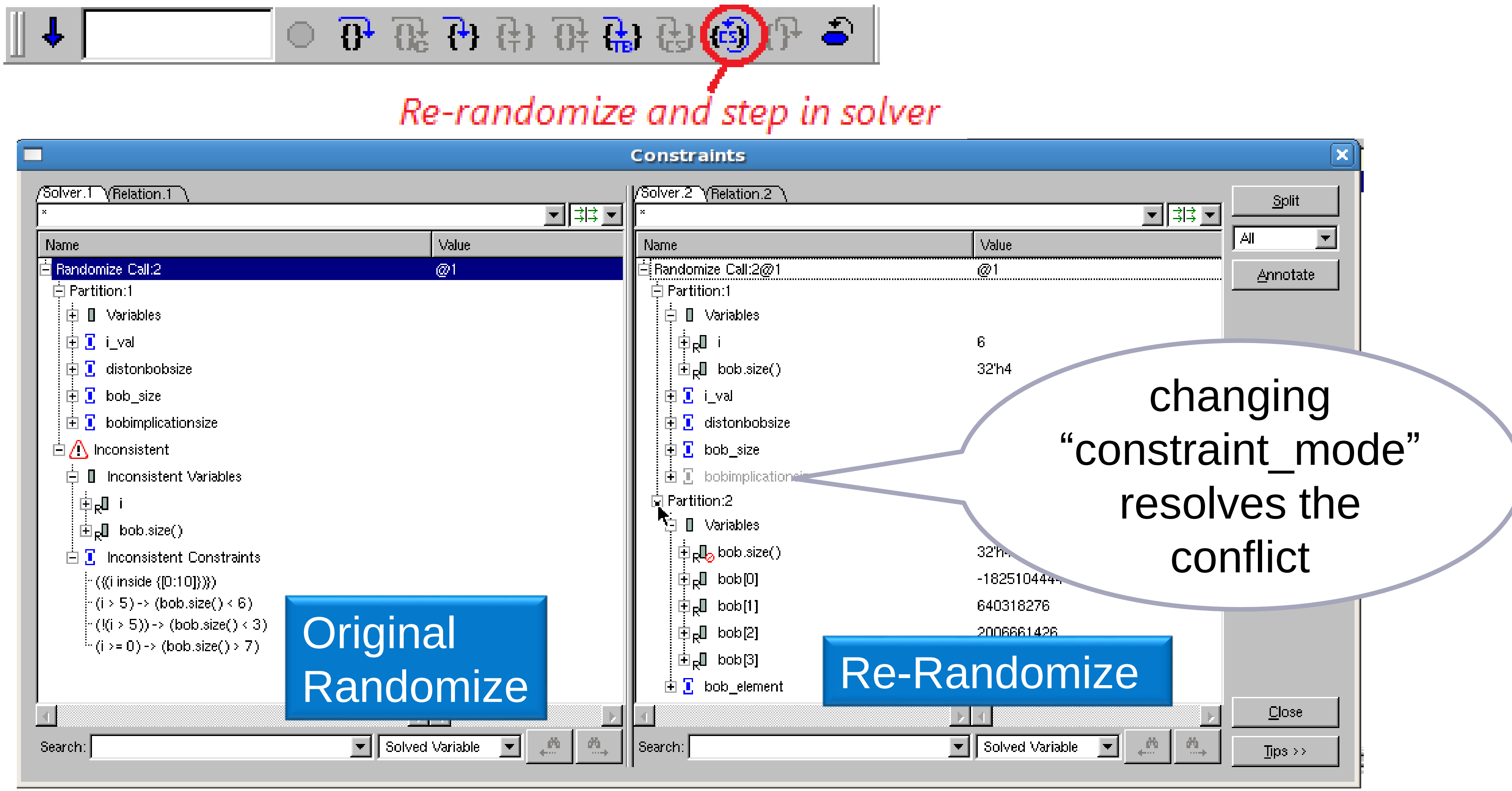



<u>Benefits</u>: User can explore and validate possible solutions without leaving DVE

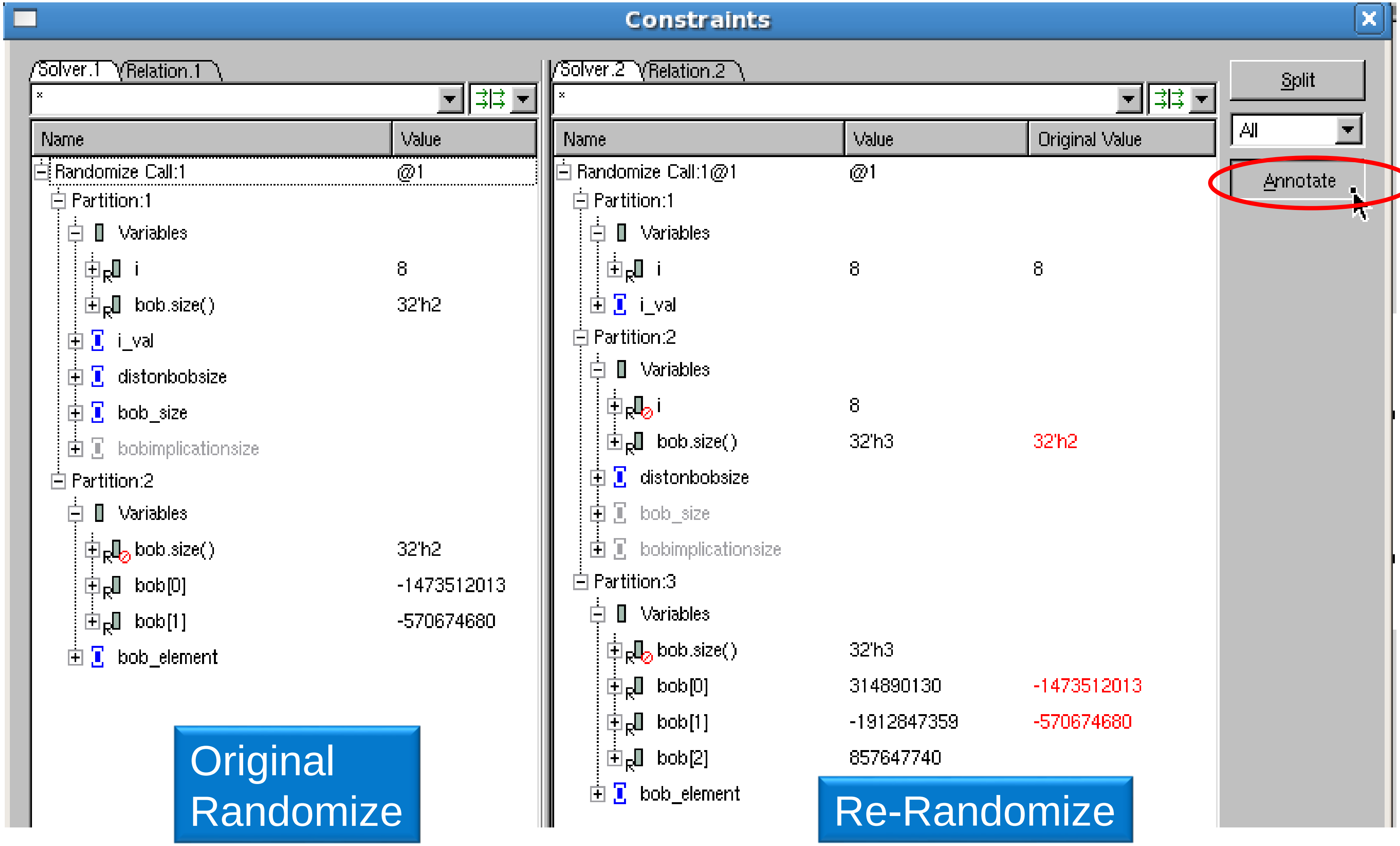
# Interactive Constraint Debug using DVE (3/5)

- **Re-randomize while stopped inside the solver**
  - after changing the states of involved variables and constraint blocks, etc.



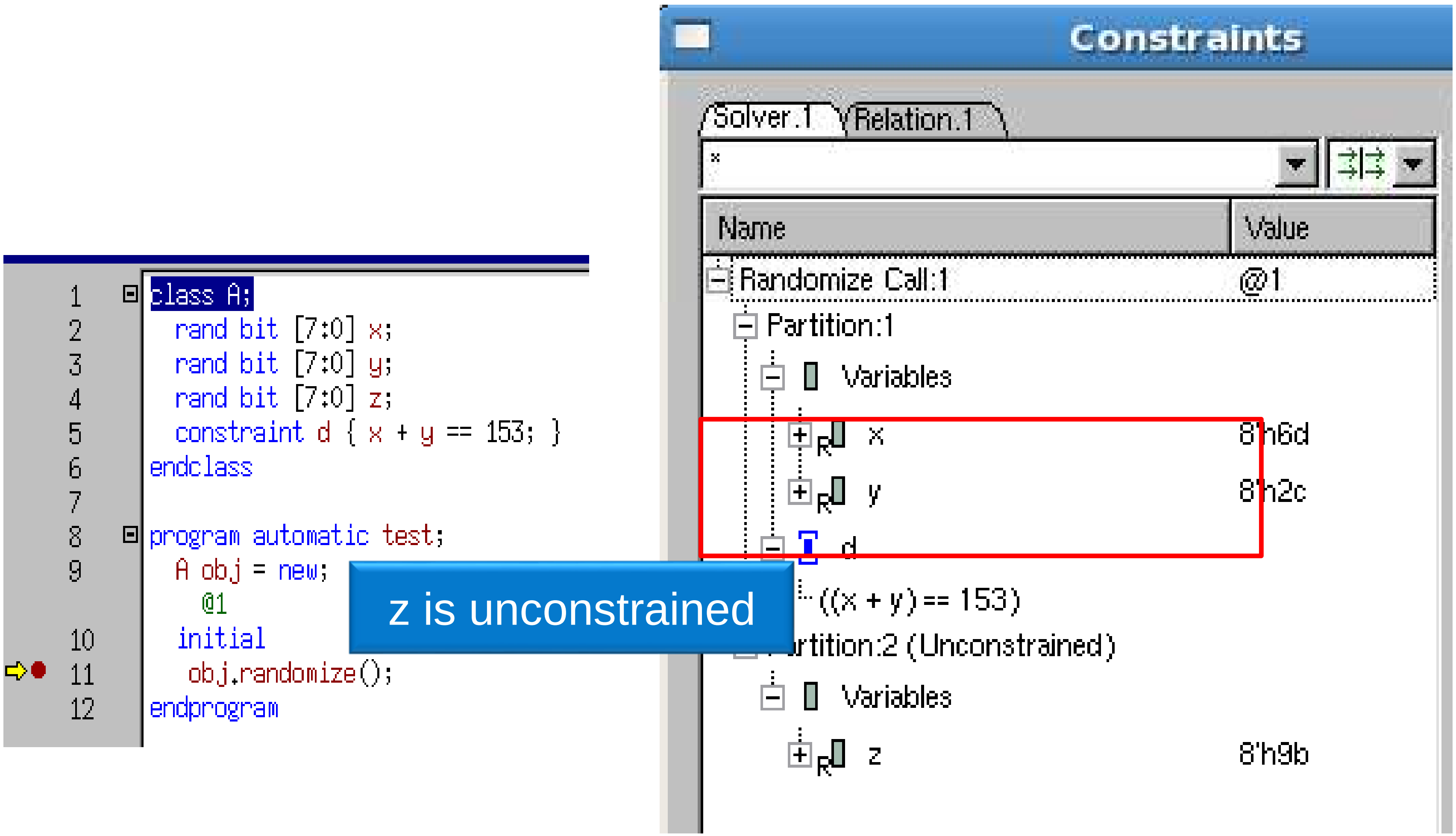
# Interactive Constraint Debug using DVE (4/5)

- Highlighting the different solved results from before



# Interactive Constraint Debug using DVE (5/5)

- **Unconstrained randomization also shown in the constraint dialog**



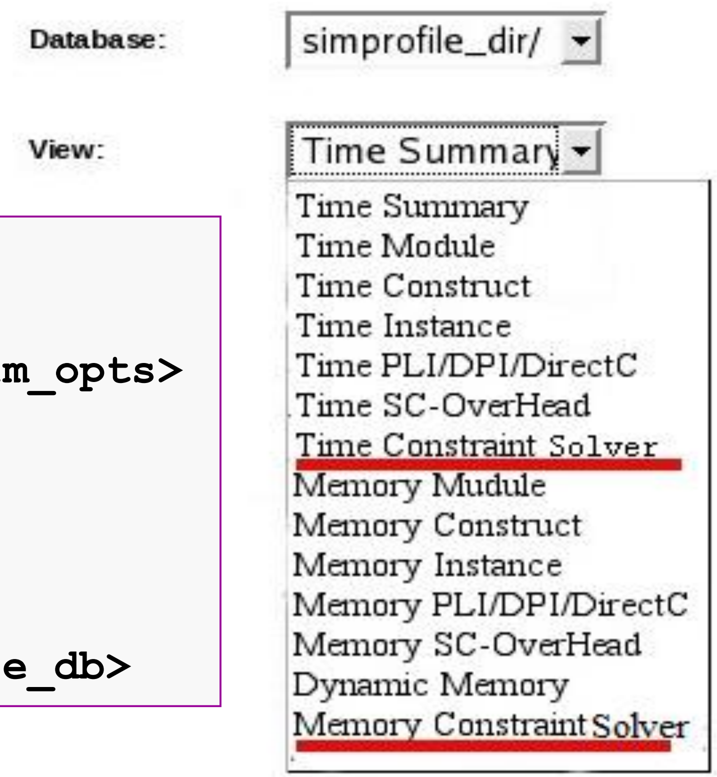
# Performance : Solver Diagnostics

- **Constraint Solver Diagnostic Report**
  - Constraint Expression Static Analysis
  - Random Variable Dynamic Analysis
  - Some constraints are mathematically hard to solve
  - Focus attention on constraints causing slowdown

```
% ./simv +ntb_enable_solver_diagnostics=<N>
            +ntb_solver_diagnostics_filename=<filename>
```

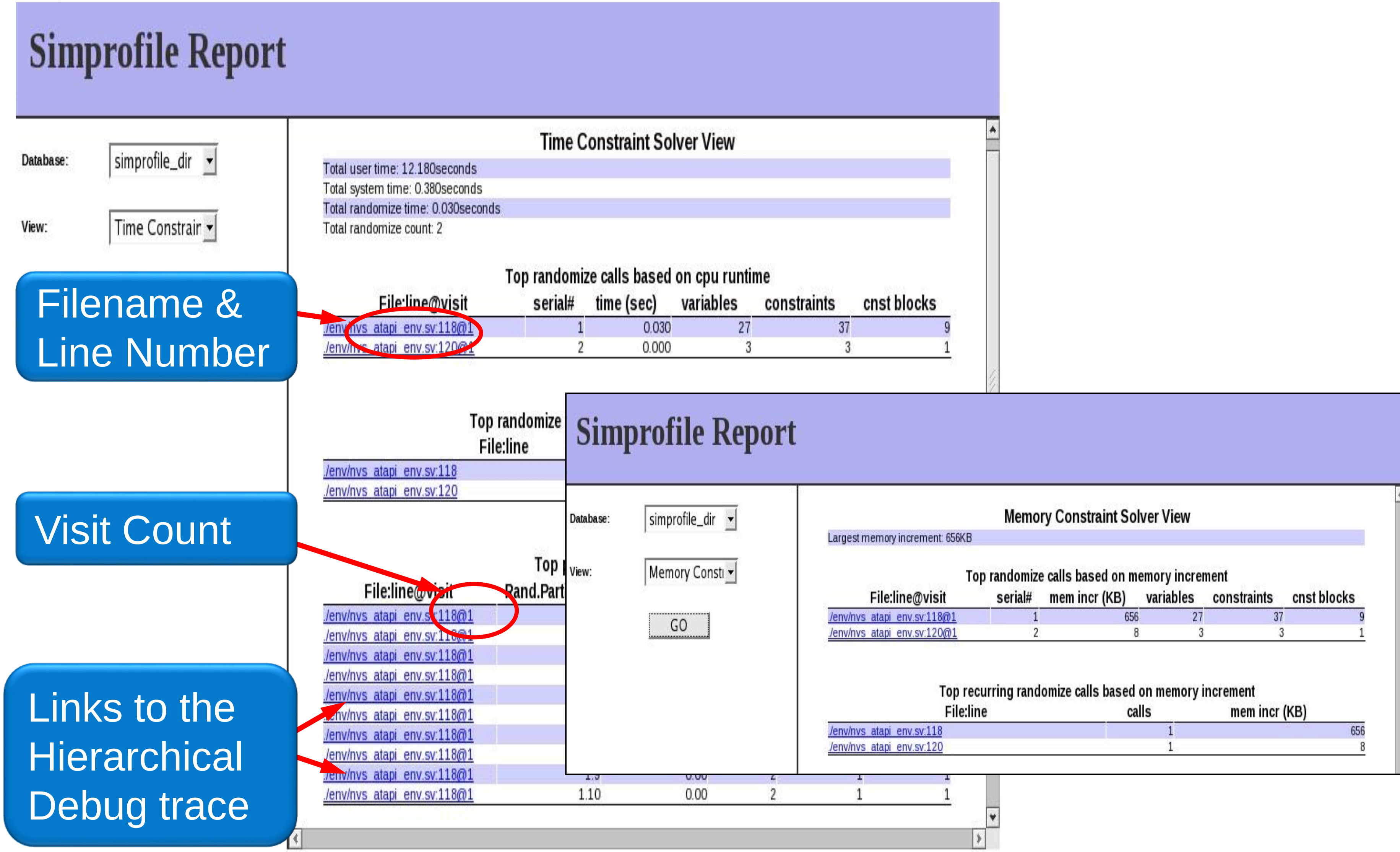
# Constraint Profile using Simprofile (1/2)

- **Use Model**
  - Starting with VCS 2012.09 constraint profiler has been integrated with vcs simprofile




```
% vcs -simprofile <vcs_opts>

% simv -simprofile [ mem | time ] <sim_opts>

% profrpt -view [time_all |
                 mem_all |
                 time_solver |
                 mem_solver ] <profile_db>
```

# Constraint Profile using Simprofile (2/2)



# Constraint Debug Methods

- **Constraint conflicts/inconsistencies**
  - VCS automatically reports/extracts the minimum sets of constraints causing inconsistencies
- **Constraints satisfied … but wrong result?**

```
% simv +ntb_solver_debug=serial
% simv +ntb_solver_debug=extract+trace
    +ntb_solver_debug_filter=<serial#>
```

- **Constraints satisfied … but too slow!**

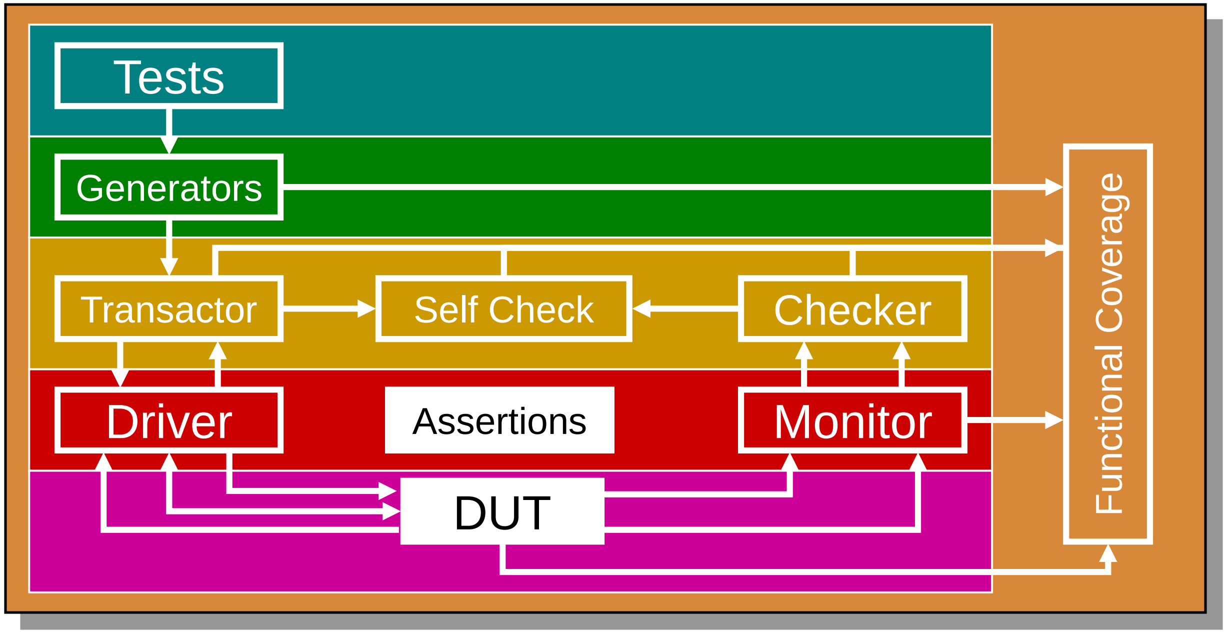
```
% simv +ntb_solver_debug=profile
% firefox simv.cst/html/profile.xml
% simv +ntb_solver_debug=extract
  +ntb_solver_debug_filter=<serial#>.<partition#>
```

  - `simv.cst/testcases contains extracted testcases`

# Methodology and Best Practices

# Best Practices : Methodology

- **Use a Well Defined Methodology (VMM/UVM)**
  - Layered testbench to enable VIP reuse
  - Name constraints consistently
  - Separate test constraints from reusable VIP



# Best Practices : State Variables

- **State Variables (not rand)**
  - Regular non-random variables
    - Or variables with **`rand_mode`** set to 0.
    - Can still be used in constraints
    - Often set once at the start of simulation
  - Example: chip configuration settings

```
class dma_config;
  bit [31:0] page_size;
endclass: dma_config
```

**Non Random**

```
class memory_transaction;
  dma_config  cfg;
  rand bit [31:0] address;
  constraint cst {
    (cfg.page_size == 256)  -> (address[6:0] == 0);
    (cfg.page_size == 512)  -> (address[7:0] == 0);
    (cfg.page_size == 1024) -> (address[8:0] == 0);
  }
endclass: memory_transaction
```

# Best Practices : Variable Ranges

- **Limit the range to be no larger than necessary**
  - Use bit variables instead of signed (**`int`**, **`byte`**)
  - Keep the size of each bit vector as small as possible

```
rand int i;                // 32 bits signed
rand bit [7:0] v;          //  8 bits unsigned
```

Smaller Range
Less Memory
Faster Runtime

- **Set values if you don't care about the result**
  - Solver can optimize constants effectively

```
rand bit [15:0] addr;
constraint dont_care { addr == 'hdead; }
```

Constant Value
Less Memory
Faster Runtime

# Performance : `pre-/post-randomize()`

- **`post_randomize()`**
  - Automatically called after **`randomize()`**
  - Procedural Code – executes much faster
  - Move arithmetic operations into **`post_randomize()`**

```
class slow;
  rand bit [7:0] data[1500];
  constraint cst { foreach(data[i]) data[i]==i; }
endclass: slow
```

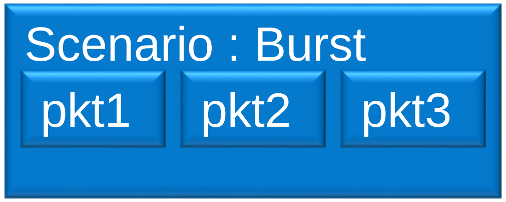
```
class fast;
  bit [7:0] data[1500];
  function void post_randomize();
    foreach(data[i]) data[i]= i;
  endfunction
endclass: fast
```

Large: 1500 Constraint

Fast: Procedural Code

# Best Practices : Randomizing Scenarios

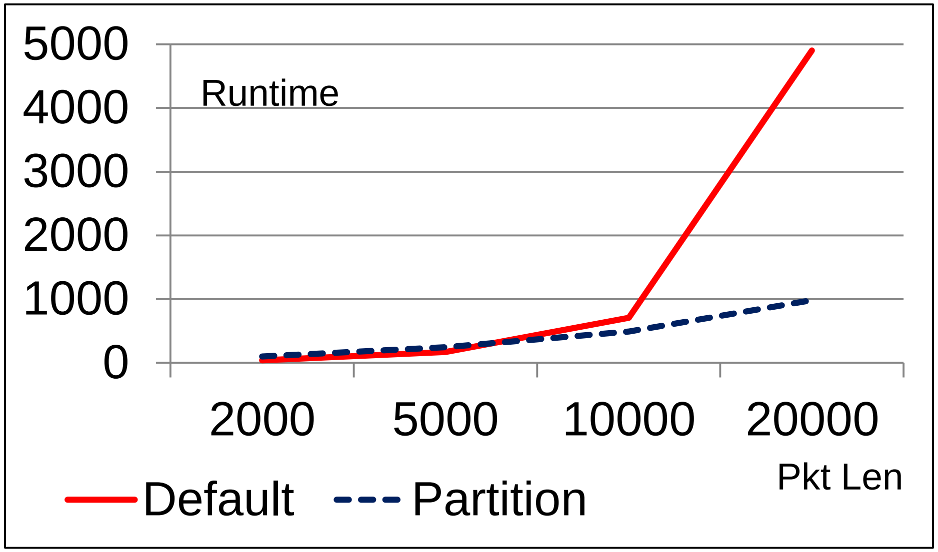
- **Scenario - Single Partition**
  - All random variables are solved at the same time
  - Flexible - allowing constraints between all items




- **Scenario - Multiple Partitions**
  - Divide into multiple randomize calls
  - Split scenario into envelope and contents
    - Identify and solve top-level control variables first
    - Randomize sub-objects with values from top-level
    - Post Randomize can also calculate values

# Performance Example

```
class packet;
  rand bit [ 7:0] id;
  rand bit [15:0] data;
endclass: packet

class burst;//Default
  rand packet pkt[16];
  constraint same_id {
    foreach (pkt[i])
      pkt[i].id
  }
endclass: burst
```




```
class burst; // Partition
  packet pkt[16];
  rand bit [7:0] id;
  function void post_randomize();
    foreach (pkt[i])
      pkt[i].randomize() with {pkt.id==id;}
  endfunction
endclass: burst
```

# SV Constraints Best Practices

- Coding Gotchas
- Solution Distribution
- Solver Performance

# Watch Out for Verilog Rules (1/4)

- **Operator Precedence**

VCS constraint solver trace (or DVE) will put extra ( )'s to make it clear what gets evaluated first

| Requirement | User writes | Simulator does |
|---|---|---|
| x is not inside a {0,1,2} | `rand int x;`<br>`constraint c {`<br>`!x inside {0, 1, 2}; }` | `(!x) inside {0,1,2}`<br>x = 2 is a valid answer! |
| if mask[i] is 1'b1, addr[i] is also 1 | `rand bit [7:0] addr, mask;`<br>`constraint c {`<br>`addr & mask == mask; }` | `addr & (mask == mask)`<br>addr is anything but 0!<br>mask can be anything! |
| k is in the (0..size) range | `rand int k;`<br>`int size = 10;`<br>`constraint c {`<br>`0 < k < size; }` | `(0 < k) < size`<br>k can be anything! |
| if x is true, y is 1; else y is 2 | `rand bit x;`<br>`rand bit [15:0] y;`<br>`constraint c {`<br>`y == x ? 1 : 2; }` | `(y == x) ? 1 : 2`<br>y can be anything! |

# Watch Out for Verilog Rules (2/4)

- **Bit length promotion**

```
class C;
   rand bit [15:0] m;
   constraint c { m== 32'hFFFF_FFFF };
endclass

C obj = new;
obj.randomize();
```

Constraint solver failure!
>> No solution <<

| | | |
|---|---|---|
| m | is | 16 bits |
| 32'hFFFF_FFFF | is | 32 bits |

Bit length promotion rule says LHS will be zero extended to 32 bits (as in RHS)

Constraint (with bit length promoted) becomes:

0000 _ _ _ _ == FFFF_FFFF

Result: Constraint solver failure due to constraint inconsistency (no solution)

# Watch Out for Verilog Rules (3/4)

- **Mixing signed and unsigned operands**

```
class A;
  rand int x;
  rand bit [3:0] y;
  constraint d {
     x > y;
     y == 3;
  }
endclass

A obj = new;
obj.randomize();
```

x (int) is **signed** 32-bit integer
y (bit [3:0]) is **unsigned** 4-bit value

Mixing signed and unsigned ➔
1. Do everything unsigned first
2. Cast to signed, as required, last

x > 32'h0000_0003, i.e.
valid x solution range: 0000_0004 .. FFFF_FFFF

Since x is signed (int)
All values with MSB=1 are negative values

x = -1624735218 is a solution

# Watch Out for Verilog Rules (4/4)

- Overflow of arithmetic operations

```
class D1;
    rand bit [31:0] a, b, c;
    constraint e {
        a + b + c == 10;
    }
endclass

D1 obj = new;
obj.randomize();
```

```
class D2;
    rand bit [31:0] a, b, c;
    constraint e {
        a + b + c == 34'd10;
    }
endclass

D2 obj = new;
obj.randomize();
```

a = 3478619480
b = 870755917
c = 4240559211

a = 8
b = 0
c = 2

bit length promotion

# Watch Out for State Variables (1/4)

- **The presence of state variables may cause constraint inconsistency, or conflicts. (i.e. no solution)**
- **constraints are checked for state variables at call to `randomize()`**
  - e.g. variable without **`rand`**

```
class A;
  int x;
  constraint c {
     x > 0;
  }
endclass

A obj = new;
obj.randomize();
```

```
==========================================
Solver failed when solving following set of constraints

integer x= 0;

constraint c // (from this) (constraint_mode = ON) (t.sv:4)
{
   (x > 0);
}
==========================================
```

x is state variable (no 'rand');
x = 0 (default for 'int')

# Watch Out for State Variables (2/4)

- `rand` **variable with** `rand_mode` **OFF becomes state variable**

```
class B;
  rand int x;
  constraint d {
     x > 0;
  }
endclass

B b = new;
b.x.rand_mode(0);
b.randomize();
```

```
========================================
Solver failed when solving following set of constraints

rand integer x = 0; // rand_mode = OFF

constraint d // (fro    is) (constraint_mode = ON) (t.sv:10)
{
   (x > 0);
}
==========                          ==
```

x is state variable
('rand' with rand_mode OFF);
x = 0 (default for 'int')

# Watch Out for State Variables (3/4)

- **Any variable not in `randomize ([args])` argument list becomes state variable**

```
class C;
  rand int x,y;
  constraint e {
     x > 0;
  }
endclass

C obj = new;
obj.randomize(y);
```

```
==========================================
Solver failed when solving following set of constraints

integer x = 0;

constraint e // (from this) (constraint_mode = ON) (t.sv:10)
{
   (x > 0);
}
==========================================
```

x is state variable
(not in randomize() arguments);
x = 0 (default for ‘int’)

# Watch Out for State Variables (4/4)

- **Function calls in constraints create partitions**
  - Return value of the function is treated as a state variable

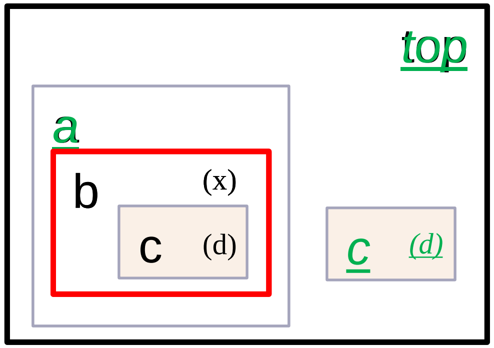
```
class A;
 rand bit [3:0] x, y, z;
 function bit [3:0]
          add (bit [3:0] m, n);
    return (m+n);
 endfunction
 constraint c {z == add (x,y);}
 constraint d {z > 7;}
endclass

A obj = new;
obj.randomize();
```

```
======================================
Solver failed when solving following set of
constraints

bit[3:0] fv_1 /*this.A::add(x, y)*/ = 4'h7;
rand bit[3:0] z; // rand_mode = ON

constraint c    // (constraint_mode = ON) (t.v:4)
{
   (z == fv_1 /*this.A::add(x, y)*/);
}
constraint d    // (constraint_mode = ON) (t.v:5)
{
   (z > 4'h7);
}
======================================
```

On some seeds this randomize may fail due to constraint inconsistency

# Watch Out for Hierarchical Constraints (1/4)

- **Effects of `rand_mode` on object**
- **Effects of object aliasing**




```
class C;
  rand bit [15:0] d;
  constraint c_d { d == 16'hbeef;}
endclass

class B;
  rand C c;
  rand int x;
  constraint c_x { x == 1234; }
endclass

class A;
  rand B b;
endclass
```

```
class Top;
   rand A a;
   rand C c;
endclass
// after constructing all objects
top.c = top.a.b.c;   //obj alias
top.a.b.rand_mode(0);
top.randomize();
```

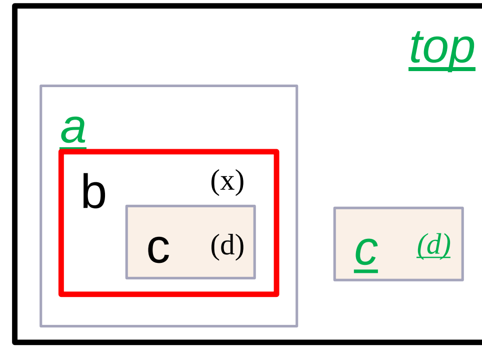
**Question:**

**top.a.b.c.d = ??**

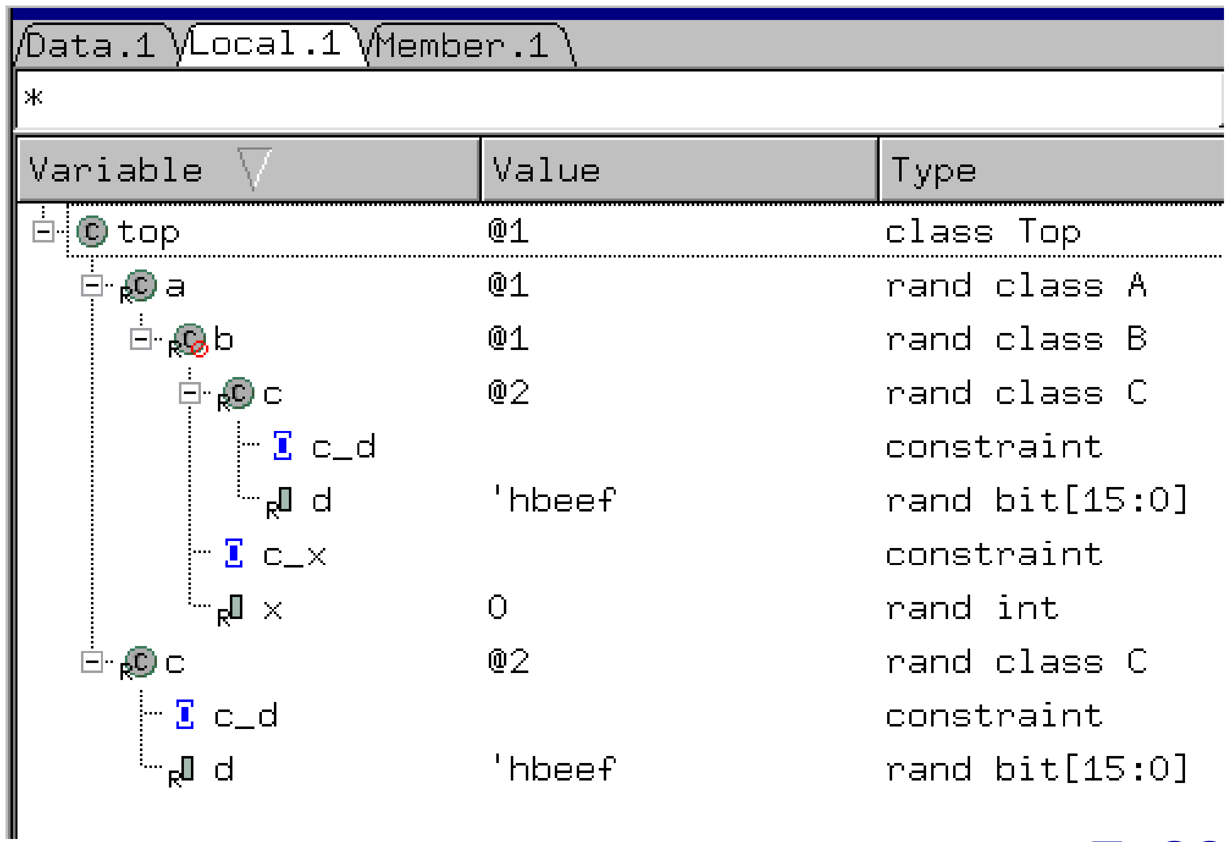
**top.a.b.x = ??**

# Watch Out for Hierarchical Constraints (2/4)

- **Effects of `rand_mode` on object**
- **Effects of object aliasing**




```
top.a.b.rand_mode(0);
top.a.b.randomize();
```




**Answer:**

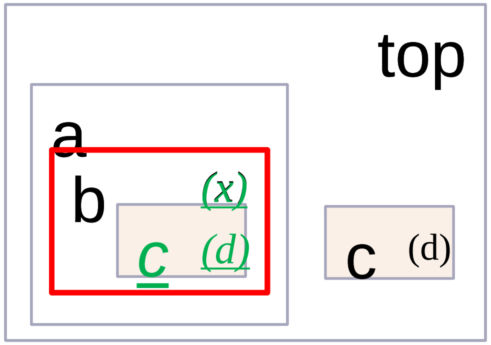
**top.a.b.c.d = ‘hbeef**

**top.a.b.x = 0**

default value for uninitialized ‘int’

# Watch Out for Hierarchical Constraints (3/4)

- **Effects of `rand_mode` on object**




```
class C;
  rand bit [15:0] d;
  constraint c_d { d == 16'hbeef;}
endclass

class B;
  rand C c;
  rand int x;
  constraint c_x { x == 1234; }
endclass

class A;
  rand B b;
endclass
```

**Question:**

top.a.b.x = ??

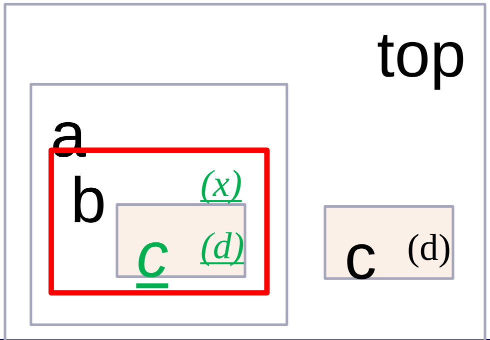
top.a.b.c.d = ??

top.c.d = ??

```
class Top;
   rand A a;
   rand C c;
endclass
// after constructing all objects
top.c = top.a.b.c;   //obj alias
top.a.b.rand_mode(0);
top.a.b.randomize();
```

# Watch Out for Hierarchical Constraints (4/4)

- **Effects of `rand_mode` on object**



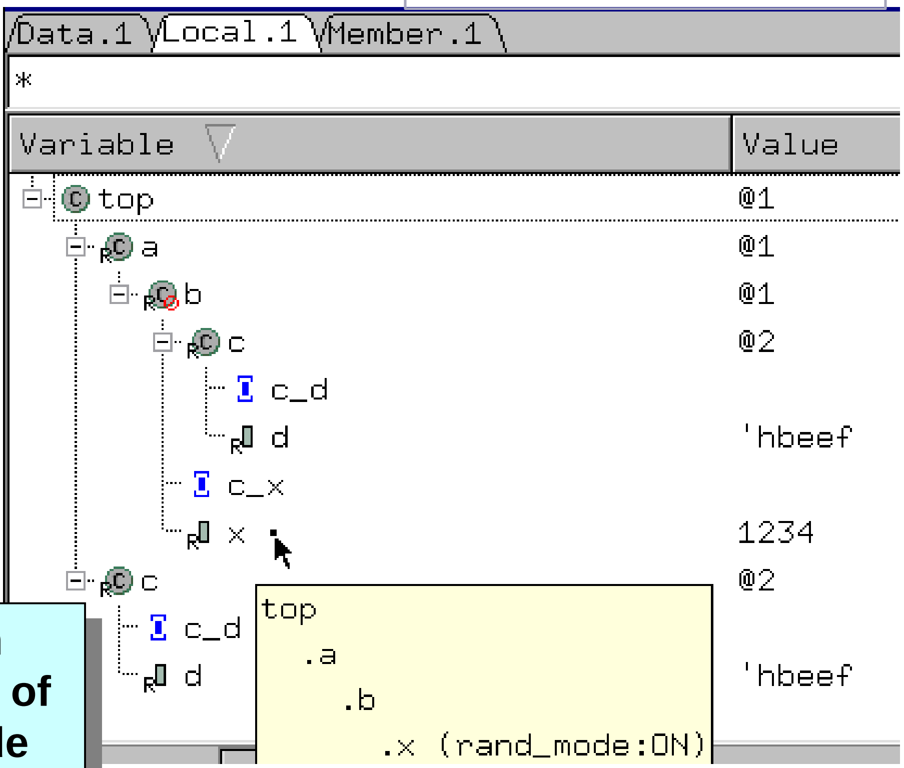

```
top.a.b.rand_mode(0);
top.a.b.randomize();
```

**Answer:**

**top.a.b.x = 1234**

**top.a.b.c.d = ‘hbeef**

**top.c.d = ‘hbeef**




**SV LRM 2012 (Section 18.8): “If the random variable is an object handle, only the mode of the handle variable is changed not the mode of the random variables within that object (see global constraints in 18.5.9)”**

# `solve..before` Changes the Probabilities

- **With uniform distribution**

```
rand bit [1:0] a, b;
constraint cst {  a >= b; }
```

| A B | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 10% | X | X | X |
| 01 | 10% | 10% | X | X |
| 10 | 10% | 10% | 10% | X |
| 11 | 10% | 10% | 10% | 10% |

- **With "solve a before b"**

```
rand bit [1:0] a, b;
constraint cst {  a >= b;
                  solve a before b; }
```

| A B | 00 | 01 | 10 | 11 | |
|---|---|---|---|---|---|
| 00 | 25% | X | X | X | } 25% |
| 01 | 12% | 12% | X | X | } 25% |
| 10 | 8% | 8% | 8% | X | } 25% |
| 11 | 6% | 6% | 6% | 6% | } 25% |

- **With "solve b before a"**

| A B | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 6% | X | X | X |
| 01 | 6% | 8% | X | X |
| 10 | 6% | 8% | 12% | X |
| 11 | 6% | 8% | 12% | 25% |
| | 25% | 25% | 25% | 25% |

```
rand bit [1:0] a, b;
constraint cst {  a >= b;
                  solve b before a; }
```

# Recommendations on `solve..before`, `dist`

## Recommendations

- First use **`dist`** to bias distribution
  - **`dist`** is more explicit and gives better control

```
rand bit [1:0] a, b;
constraint cst {
 a >= b;
 b dist {
   0 := 15, 1 := 30, 2 := 5, 3 := 50
 };
}
```

| A B | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 4% | X | X | X |
| 01 | 4% | 10% | X | X |
| 10 | 4% | 10% | 3% | X |
| 11 | 3% | 10% | 2% | 50% |
| | **15%** | **30%** | **5%** | **50%** |

- If this distribution is not satisfactory, analyze the problem and see if **`solve-before`** helps

# Implementation Choices

- **Given the same design requirement, there are sometimes more than one ways to describe it using constraints**
- **Understand their impact**
  - Readability
  - Runtime Performance
  - Use model differences for the end users
- **VCS constraint solver is continuously enhanced**
  - Performance may have improved in newer VCS releases

# Implementation Choices: Example 1

- **2048-bit addr & mask vectors**
  - If ***mask[i]*** is 1, ***addr[i]*** shall be 1, else ***addr[i]*** is don't-care

```
class cpu_trans;
  rand bit [2047:0] addr;
  rand bit [2047:0] mask;
  constraint slower {
    foreach (mask[i])
      if (mask[i] == 1) addr[i] == 1;
  }
endclass
```

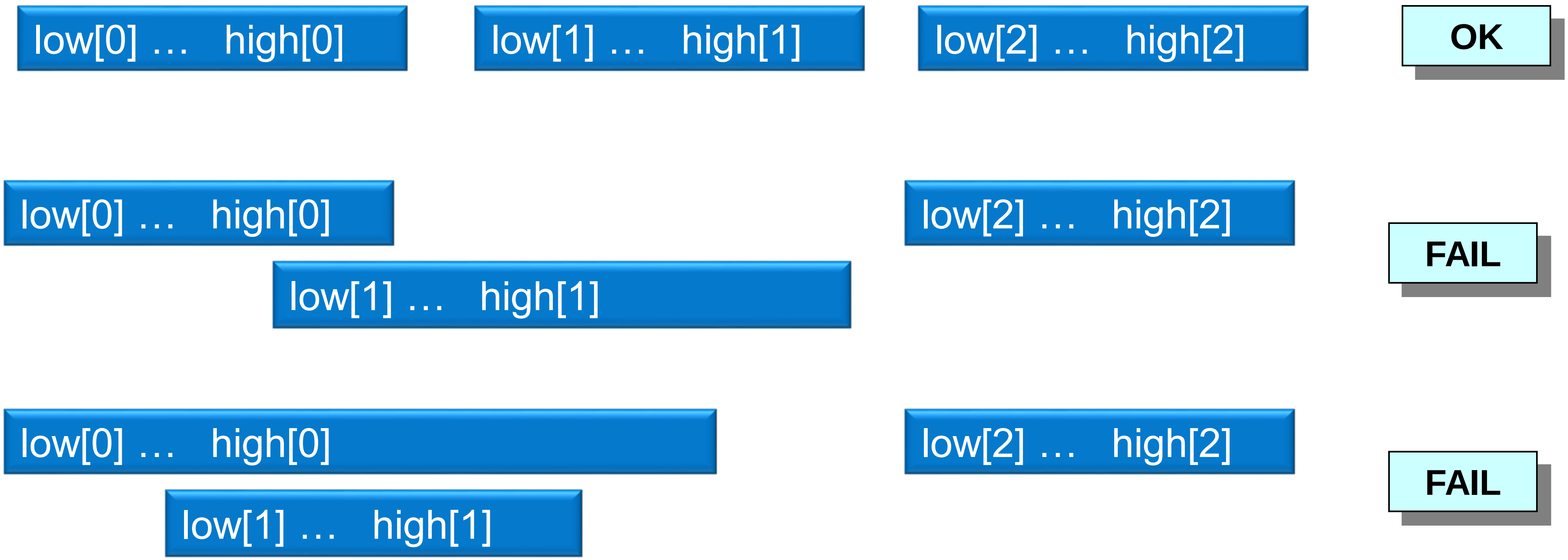
| VCS | CPU Time |
|---|---|
| 2011.12-2 | 4.8 seconds |

```
class cpu_trans;
  rand bit [2047:0] addr;
  rand bit [2047:0] mask;
  constraint faster {
      (addr & mask) == mask;
  }
endclass
```

| VCS | CPU Time |
|---|---|
| 2011.12-2 | 0.48 seconds |

# Implementation Choices: Example 2 (1/3)

- **Need to generate random non-overlapping ranges**



# Implementation Choices: Example 2 (2/3)

- **Start and end are not inside any other ranges**

```
class mem;
  rand bit [31:0] low[$]
  rand bit [31:0] high[$];
  constraint cst {
    foreach (low[i]) {
      low[i] < high[i];
      foreach (low[j]) {
        (i != j) -> !(low[i] inside {[ low[j] : high[j] ]});
        (i != j) -> !(high[i] inside {[ low[j] : high[j] ]});
      }
    }
  }
endclass
```

low[k] … high[k]

low[p] … high[p]

low[m] … high[m]

**Problem : Number of constraints order(n**2)**

**Does not scale as number of ranges increases**

# Implementation Choices: Example 2 (3/3)

- **Start and end are in order, and increasing**

```
class mem;
  rand bit [31:0] low[$]
  rand bit [31:0] high[$];
  constraint cst {
    foreach (low[i]) {
      (i > 0) -> (low[i] < high[i];
      (i > 0) -> (low[i] > high[i-1]);
    }
  }
endclass
```

low[0] … high[0]

low[1] … high[1]

low[2] … high[2]

**Better: Number of constraints order(n)**

**Scales with number of ranges**