The Top Most Common SystemVerilog Constrained Random Gotchas

Author: Ahmed Yehia

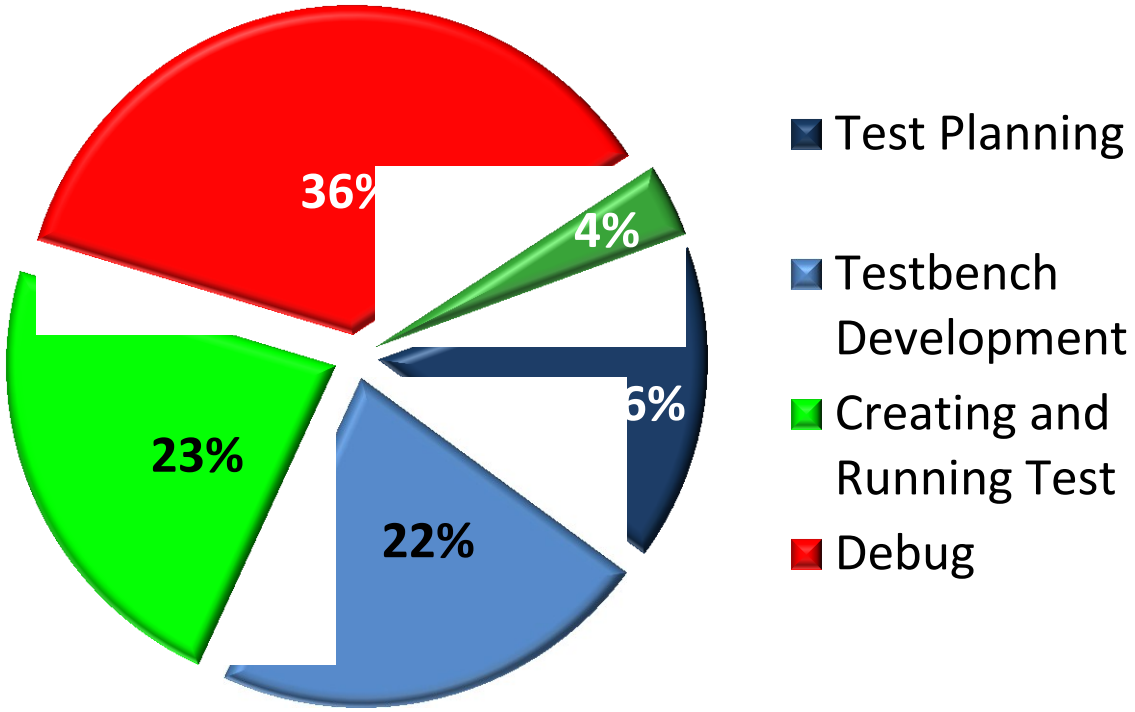
Presenter: Gabriel Chidolue

accellera SYSTEMS INITIATIVE

2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Motivation

- More time is taken in debug than any other project task
- Time wasted in debugging constrained random related problems is significant



Wilson Research Group and Mentor Graphics, 2012 Functional Verification Study, Used with permission

Contribution

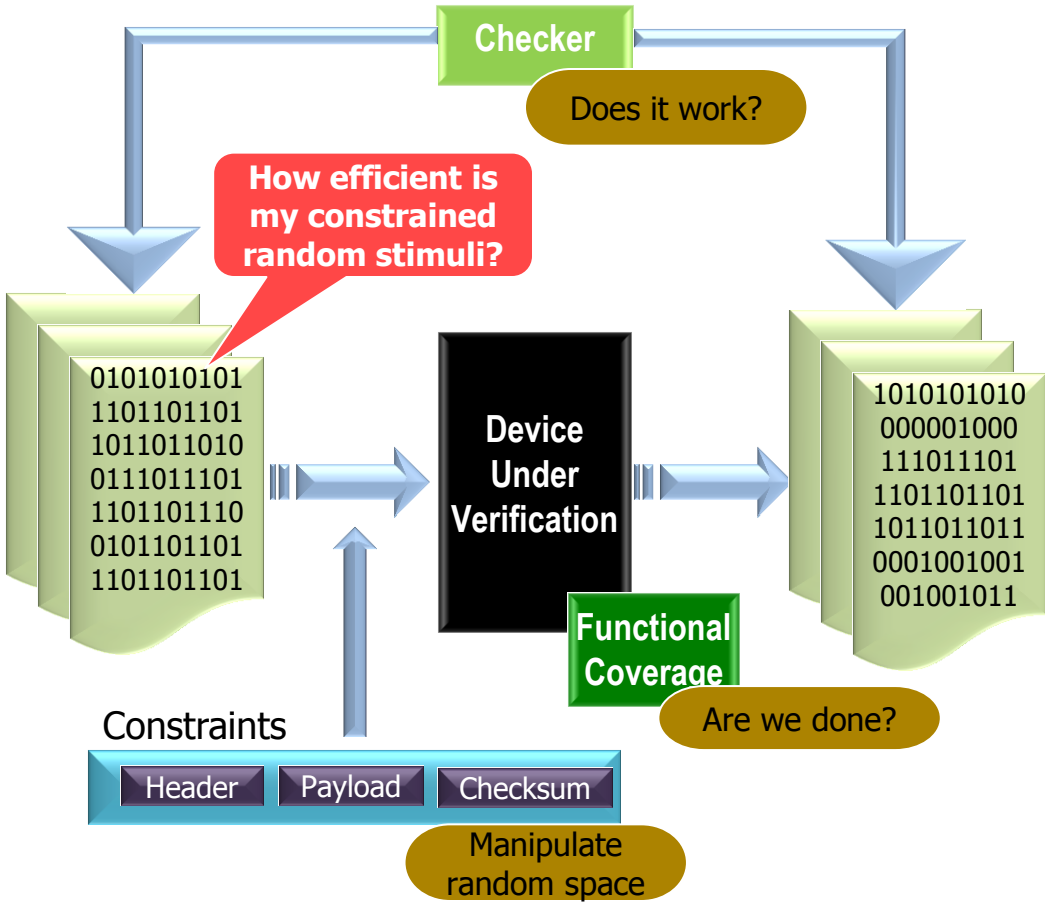
- Illustrates top most common SystemVerilog and UVM constrained random gotchas
- Helps
  - Eliminate/reduce unnecessary debug times when encountering randomization failures
  - Eliminate/reduce unexpected randomization results
  - Eliminate/reduce code random instability
  - Ensure efficiency when coding random constraints

Outline

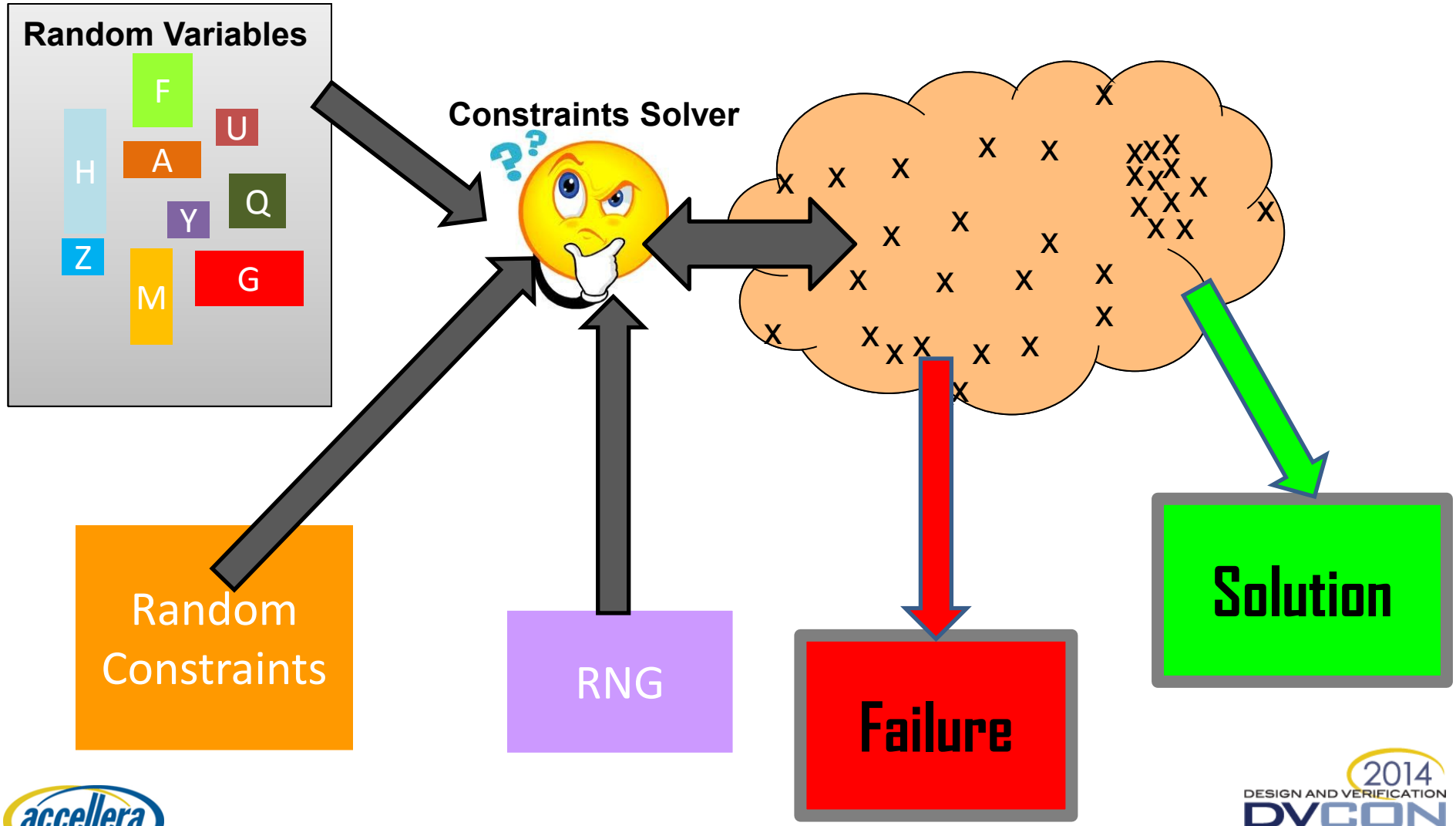
- Introduction to Constrained Random Verification
- Randomization Failures Gotchas
- Randomization Results Gotchas
- Randomization Runtime Performance Gotchas

Constrained Random Verification

- Defines stimulus at high level of abstraction (random space)
  - Random variables and their range
  - System Constraints
- More efficient than directed tests
  - Usually complemented by directed tests to close coverage
- Requires a measurement strategy to assess the verification progress
  - Metric Driven Verification



Introduction to SystemVerilog Constrained Random



Introduction to SystemVerilog Constrained Random

SystemVerilog Randomization Methods

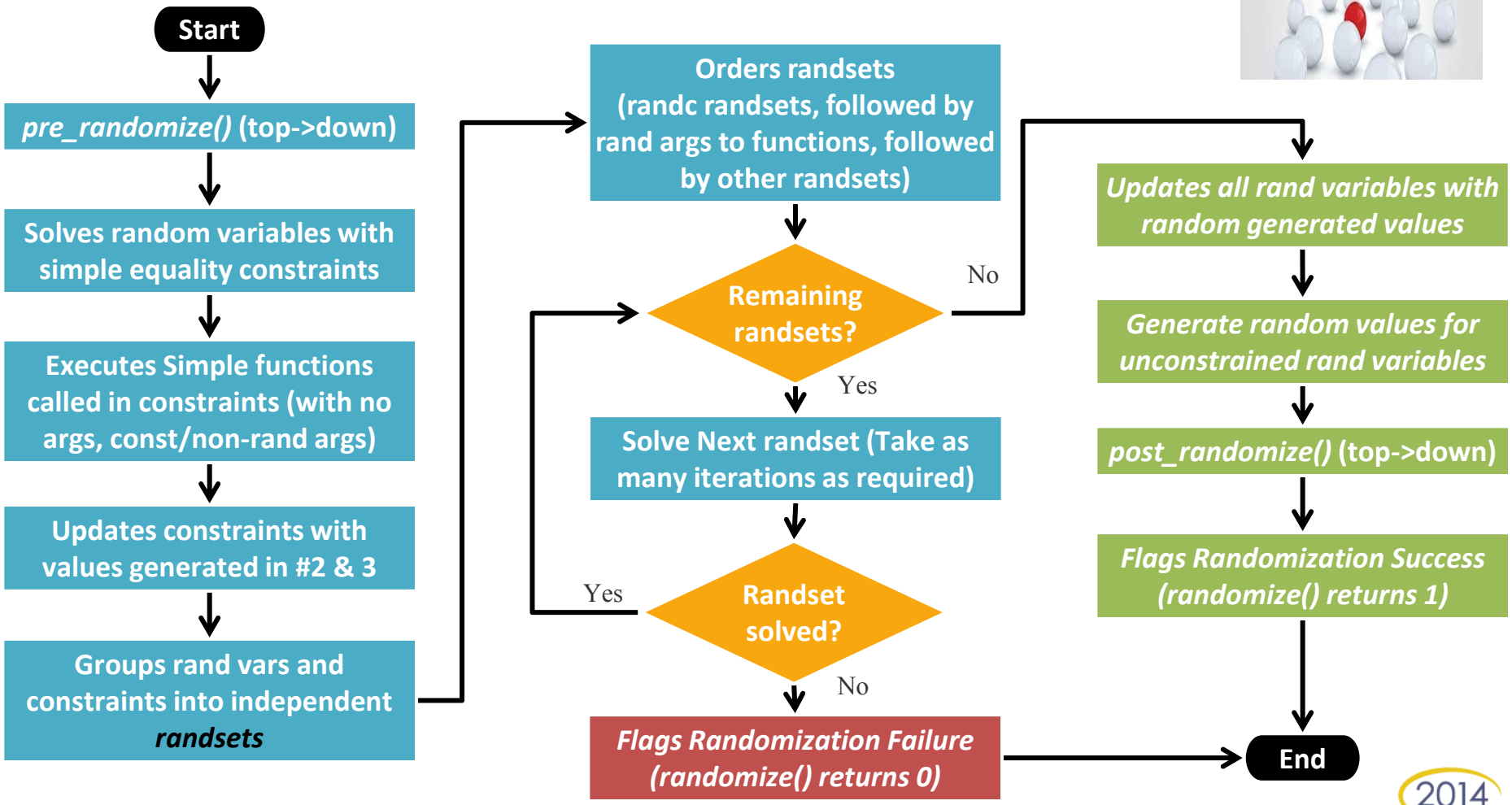
- **randomize()**
  - Built-in class method
  - Randomizes class fields with *rand/randc* qualifiers according to predefined constraints
  - Accepts inline constraints using the "*with*" clause
  - can be called to recursively randomize all random variables of a class, or to randomize specific variable(s)
- **$urandom()**
  - Called in a procedural context to generate a pseudo-random number
- **$urandom_range()**
  - Returns an unsigned random integer value within a specified range
- **std::randomize()**
  - Can be called outside the class scope to randomize non-class members.
  - Can accept inline constraints using the "*with*" clause.

SystemVerilog Randomization Constraints

- Expressions need to be held true by the Solver when solving a randomization problem
- May include random variables, non-random state variables, operators, distributions, literals, and constants
- Can be *hard* (default) or *soft*
- Can be switched on/off using *constraint_mode()*
- special operators
  - *inside* (set membership),
  - -> (implication),
  - *dist* (distribution/weighting),
  - *foreach* (iteration),
  - *if..*else (conditional),
  - and *solve..before* (probability and distribution)
  - *unique*,…

Constraints Solver

How it works ?(obj.randomize())



Outline

- Introduction to Constrained Random Verification
- **Randomization Failures Gotchas**
- Randomization Results Gotchas
- Runtime Performance Gotchas

My randomization attempt failed and I was not notified !

```
class instr_burst;
  rand bit [15:0] addr, start_addr, end_addr;
  rand bit [3:0]  len;
  constraint addr_range {addr >= start_addr; addr <= end_addr - len;}
endclass
instr_burst i1 = new;
i1.randomize() with {start_addr != 0; end_addr == 16'h0008; len == 4'h8;};
```

What happens when result of *randomize()* is NOT captured?

```
if (! i1.randomize())
  $error ("Randomization of object c1 failed!");
```

```
assert(i1.randomize());
```

Always capture *randomize()* result to avoid implicit *void casting*

I am only randomizing a single variable in a class, yet I am encountering a randomization failure!

```
class trans;
  rand bit [7:0] a, b, c ;
  constraint constr { b < a; }
endclass
initial begin
  trans t1 = new;
  assert (t1.randomize (b)); //Randomization failure!
end
```

All constraints still need to be satisfied!

```
trans t1 = new;
assert (t1.randomize);
assert (t1.randomize (b));
```

- Always issue at least one full *randomize()* before selected variables *randomize()*
- Avoid (as possible) assigning other *rand* variables manually without solver jurisdiction

I am encountering cyclic dependency errors between random variables!

```
class instr;
  rand bit [7:0] a, b, c ;
  constraint prob{solve a before b;
                  solve b before c;
                  solve c before a;}
endclass
```

```
class instr;
  rand bit [7:0] a ;
  constraint c { a[0] == foo (a) ;}
endclass
```

What is the probability of randomization success?

8-bits of "a" are solved first before solver attempt to satisfy the equality!

- Random variables passed as function arguments are forced to be solved first by the Solver
- Solver does not look into functions contents

```
function void post_randomize();
  a[0] = foo (a);
endfunction
```

I am encountering cyclic dependency errors between randc variables!

```
class instr;
  randc bit [7:0] a ;
  randc bit [3:0] b ;
  constraint c { a == b; }
endclass
instr i = new;
assert(i.randomize());
```

Are "a" and "b" evaluated together or separately? If evaluated separately, what is the order of evaluation?

```
constraint c { a[3:0] == b; }
```

What if "b" was *rand*?

- *randc* cycles operate on single variables
- *randc* variables are evaluated separately
- Beware equality between unmatched size *randc* variables
- Because of this, cyclic nature of randc variables can even be compromised!

I am getting randomization failures when using array.sum()/array.product() reduction methods in constraints!

```
class trans;
  rand bit descr [];
  constraint c {
    descr.sum()  == 50;
    descr.size() == 100;
  }
endclass
```

What is the width/precision of sum() result?

```
constraint c {
   descr.sum() with (int'(item)) == 50;
   descr.size() == 100;
 }
```

- sum()/product() results are computed with a width/precision of array base type
- Explicitly cast the array element (i.e. *item*) to an *int* data type when needed

Guidelines Summary

Randomization Failures Gotchas

- Always capture randomize() result
- Beware base array type when using sum()/product()
- Beware dependencies between variables cannot be solved together
  - Randc variables
  - Random variables passed as functions arguments
- Randomizing single var in cluster, does NOT mean Solver will abandon all other constraints

Outline

- Introduction to Constrained Random Verification
- Randomization Failures Gotchas
- **Randomization Results Gotchas**
- Randomization Runtime Performance Gotchas

Random values generated change from run to run; I could not reproduce a test failure or validate a fix!

```
virtual task body;
  random_seq rand_s = new;          //Line A
  simple_seq rw_s   = new;
  fork begin
      assert (rw_s.randomize()); //Line B
      rw_s.start();
  end
    ...
  join
endtask
```

✗

Does line "A" affect the random stability of line "B"?

- *Randomize() results depends on previous state of RNG*
- *Object initial RNG depends on parent's RNG, while it changes state after each randomize()*

```
static int global_seed = $urandom; //Static global seed
...
fork begin
  rw_s.srandom(global_seed + "rw_s"); //Reseed sequence
  assert (rw_s.randomize());
  rw_s.start();
end
```

✓

- Use Manual seeding to manually set the RNG of a given thread or an object to a specific known state
- In UVM, components are re-seeded during their construction based on their type and full path names, while sequences are re-seeded automatically before their actual start.

!

My inline constraints are not applied

```
class trans;
  rand bit [31:0] addr;
endclass
class seq;
  rand bit [31:0] addr;
  trans t;
  assert(t.randomize() with {t.addr == addr;});
endclass
```

What is the randomization result w.r.t. the constraint intent?

Equivalent Constraint

```
assert(t.randomize() with {t.addr == t.addr;});
```

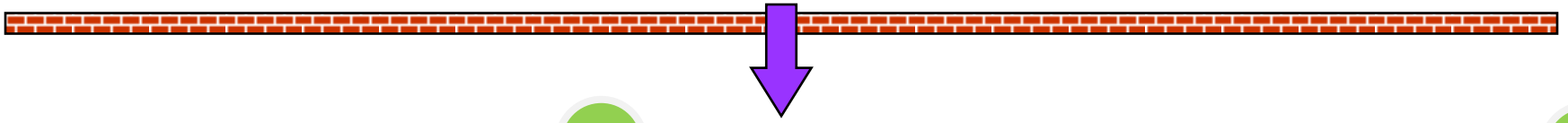
```
assert(t.randomize() with { addr == local::addr; });
```

Will '*this*' work?

- *Unqualified names are resolved by searching first in the scope of the randomize() **with** object, followed by a search in the local scope.*
- *The **local::** qualifier modifies the resolution search order.*

My foreign language random generation is not affected by the initial simulation seed change

- Normally, the initial SystemVerilog simulation seed, does not affect foreign language code
- This can be resolved by passing the simulation initial seed to the foreign language code



```
// C/C++ side
static int sim_seed;
void set_foreign_seed(int seed){
  sim_seed = seed;
}
int stimgen () {
  int desc;
  ...
  srand(sim_seed);
  desc = rand();
  ...
  return 0;
}
```

DPI/PLI

```
// SystemVerilog side
import "DPI-C" context function void
                    set_foreign_seed(int seed);
int global_seed = $urandom;
initial
  set_foreign_seed (global_seed);
```

Unexpected negative values are generated upon randomize!

```
class trans;
  rand int addr;
  constraint c {
    addr < MAX_ADDR;
  }
endclass
class seq;
  rand bit [31:0] addr;
  trans t;
endclass
```

int, byte, and variables declared as signed can hold negative random values

```
  rand bit [31:0] start_addr;
  rand bit [5:0] length;
  bit [31:0] start_end_addr_hash [bit[31:0]];
  constraint c {   //Generate Non-Overlapping address ranges
    if (start_end_addr_hash.num()) {
      foreach (start_end_addr_hash [i]) {
        !(start_addr inside {[i-length+1 :
                              start_end_addr_hash [i]]});
      }
    }
    lengt == 6'h10;
  }...
start_ addr_hash [start_addr] = start_addr + length - 1;
```

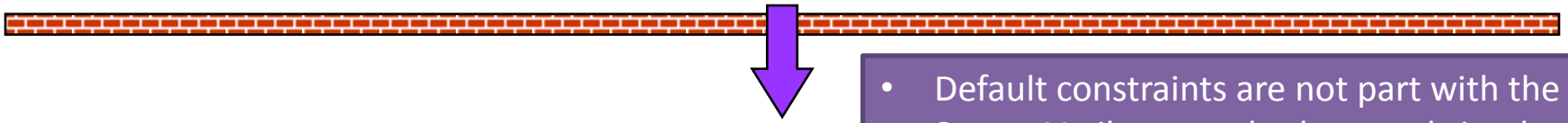
What happens when "start_addr" of a previous randomization attempt was picked to be smaller than the address range "length"?

```
foreach (start_end_addr_hash [i]) {
  if (i >= length ) {
    !(start_addr inside {
        [i-length+1 :
         start_end_addr_hash [i]]});
  } else {
    !(start_addr inside {
      [0 : start_end_addr_hash [i]]});
  }
}
```

I am getting unexpected random results when using default constraints

```
default constraint c1 {x < 10; y > z;}
...
constraint c2 {x < 5;}
```

What is the probability of getting "z" be greater than "y"?



```
constraint c1 {soft x < 10; y > z;}
constraint c2 {x < 5;}
```

- Default constraints are not part with the SystemVerilog standard; several simulators allow them for legacy purposes
- Once any variable used in default constrained is used in another constraint , the entire default constraint is ignored
- **<u>Do NOT use default constraints</u>**; use *soft* constraints instead.

Guidelines Summary

Randomization Results Gotchas

- Study SystemVerilog random stability; make use of manual seeding and/or UVM.
- Beware the signed nature of the output.
- Results from randc may not be always cyclic.
- Foreign language random code does NOT implicitly follow initial SystemVerilog seed.
- Do NOT use *default* constraints.

Outline

- Introduction to Constrained Random Verification
- Randomization Failures Gotchas
- Randomization Results Gotchas
- **Randomization Runtime Performance Gotchas**

Writing Efficient Constraints

- Often users write constraints focusing on functionality and not performance
- Surprises at runtime w.r.t. Solver overhead

```
//Dynamic array with unique elements
class trans;
rand bit [31:0] addrs [];
constraint unique_arr{
  foreach (addrs[i])
    foreach (addrs[j])
      if (j < i)
        addrs [i] != addrs [j];
}
endclass
```

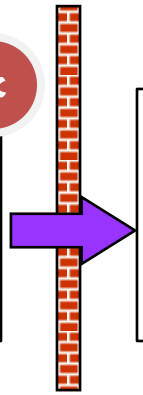
Foreach is performance greedy!

```
class dummy_c;
  randc bit [31:0] val;
endclass
class trans;
  rand bit [31:0] addrs [];
  function void post_randomize();
     dummy_c dc = new;
    foreach (addrs[i]) begin
      assert (dc.randomize);
      addrs[i] = dc.val;
    end
  endfunction
endclass
```

Writing Efficient Constraints (cont.)

Avoid complex operations that overburden the Solver

```
constraint arith {
  b == a * 64;
  d == c / 8;
  f == 2 ** e;
  addr % 4 == 0;
}
```

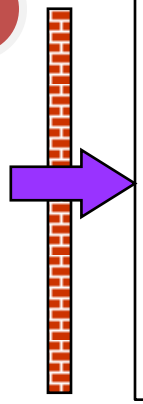


```
constraint arith {
  b == a << 6;
  d == c >> 3;
  f == 1 << e;
  addr [1:0] == 0;
}
```

Bitwise operations are relatively easy to solve even with a BDD engine

dist operators can have runtime overhead. Reduce usage as possible

```
rand bit [4:0] rd;
constraint mode_32bit{
  mode == 32 -> rd < 16;}
constraint rd_dist {
  mode == 32 -> rd [3:0] dist
{
     [4'b0000:4'b1110]: 3,
      4'b1111         : 1
  };
}
```



```
rand bit [4:0] rd;
rand bit [1:0] rd_eq_15;
constraint mode_32bit{
  mode == 32 -> rd < 15;
}
function void post_randomize();
 if (mode == 32)
   if (rd_eq_15 == 2'b11)
     rd = 5'b01111;
endfunction
```

Guidelines Summary

Performance Gotchas

- Minimize number of rand variables
- Use optimal data types
- Replace simple constraints with $urandom() calls
- Avoid complex operators/operations
- Do NOT use randc modifiers unless really needed
- Move constraints to pre/post_randomize() when possible
- Avoid using foreach clause in constraints unless really needed
- Beware the solve-before misconception
- Apply late randomization when applicable

References

- Mentor Graphics Verification Academy, www.verificationacademy.com
- IEEE Standard for SystemVerilog, *Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2012, 2012
- UVM User Manual, uvmworld.org.
- UVM Random Stability: Don't leave it to chance, Avidan Efody, DVCon 2012.
- Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them, Stuart Sutherland and Don Mills, Springer

Questions