2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

The Top Most Common SystemVerilog Constrained Random Gotchas

Ahmed Yehia, Mentor Graphics Corp., Cairo, Egypt, ahmed_yehia@mentor.com

***Abstract*—The Constrained Random (CR) portion in any verification environment is a significant contributor to both the coding effort and the simulation overhead. Often, verification engineers waste a significant amount of time debugging problems related to CR in their SystemVerilog[1], and UVM[4], testbenches. The paper illustrates the top most common SystemVerilog CR gotchas, which when carefully studied and addressed would help decrease debug times related to CR, reduce random instabilities, and boost productivity.**

Keywords—SystemVerilog; UVM; Constrained Random; Random distribution; Random Stability

I. INTRODUCTION

Over the years, Constrained Random Verification (CRV) became the market focus. CRV, in its most ideal form, is seen as an effective way in improving the verification process; it is easier to build a single Constrained Random (CR) test that is equivalent to many directed tests (despite the fact that building a CRV environment would be more complex than its Directed counterpart would be). However, CRV cannot be used in a standalone manner; it needs to go hand-in-hand with a measurement strategy to assess the Design Under Verification (DUV) verification progress. Here, the term Coverage Driven Verification (CDV)[1] arose, in which a coverage model that represents different features of the DUV to be verified is built, and coverage is collected during CR tests run. Although CR stimuli provide a tremendous value towards faster functional coverage closure over directed tests, yet modeling and debugging constraints, and random stimuli, is not trivial and suffer many challenges.

In programing, a "gotcha" is a documented language feature, which, if missed, causes unexpected or unintuitive behavior. This paper illustrates the top most common SystemVerilog and UVM constrained random gotchas, which when carefully studied and addressed would help: 1) eliminate/reduce unnecessary debug times when encountering unexpected randomization failures, 2) eliminate/reduce unexpected randomization results, 3) eliminate/reduce random instability, and 4) ensure efficiency when coding random constraints.

II. SYSTEMVERILOG CONSTRAINED RANDOM OVERVIEW

This section provides a very basic knowledge about the SystemVerilog constrained random[2]. A given randomization problem consists of a set of random variables and a set of randomization constraints. A constraint solver is the engine attempting to solve the randomization problem at hand following pre-known steps.

A. *SystemVerilog randomization methods*

The SystemVerilog language provides multiple methods to generate and manipulate random data:

- $urandom(): System function can be called in a procedural context to generate a pseudo-random number.
- $urandom_range(): System function returns an unsigned random integer value within a specified range.
- randomize(): Built-in class method used to randomize class fields with *rand/randc* qualifiers according to predefined constraints. It can accept inline constraints using the "*with*" clause in addition to the constraints defined in a class context. It can be called to recursively randomize all random variables of a class, or to randomize specific variable(s) as well (either defined with the *rand* qualifier or not), keeping all pre-defined constraints satisfiable.
- std::randomize(): Can be called outside the class scope to randomize non-class members. Can accept inline constraints using the "*with*" clause.

[1] Metric Driven Verification is a more general term.
[2] For more detailed information, refer to the IEEE Std P1800™-2012, *IEEE Standard for SystemVerilog language[1]*.

The IEEE 1364 Verilog language provided other randomization system functions like *$random* and *$dist_**, these functions should not be used in SystemVerilog as they are not part of the random stability model.

B. SystemVerilog randomization constraints

- Constraints are expressions that need to be held true by the constraint solver when solving a randomization problem. Constraint expressions may include random variables, non-random state variables, operators, distributions, literals, and constants.
- Constraints can be defined as explicit properties of a class, or specified in-line with *randomize()* calls.
- Constraints can be *hard* (default) or *soft* according to their declaration.
- Constraints special operators: *inside* (set membership), -> (implication), *dist* (distribution/weighting), *foreach* (iteration), *if..*else (conditional), and *solve..before* (probability and distribution).
- Constraints can be switched on/off using the *constraint_mode(1)*/*constraint_mode(0)* built-in method.

C. How does the constraint Solver work?

In an attempt to solve a specific randomization problem, the Solver takes the following steps when it encounters a *randomize()* call as dictated by the SystemVerilog LRM[1]:

1. Calls the *pre_randomize()* virtual function recursively in a top-down manner.
2. Scans the entire randomization cluster enclosing all random variables and constraints.
3. Solves random variables with simple equality constraints (e.g. constraint c {x ==5;};).
4. Executes simple functions called in constraints; functions with no arguments or whose arguments are constants or variables that do not belong to the current randomization cluster. Remember that the Solver does not look into functions' contents, and so even if functions access random variables in their body, they are still going to be called and evaluated substituting random variables with their current values.
5. Updates constraints of the current randomization cluster by substituting with values deduced in steps #3 and #4 (also non-random variables used in constraints are substituted with their current values).
6. Groups random variables and constraints into independent *randsets*. A *randset* holds random variables that share common constraints; i.e. variables that their solution depends on each other because of common constraints, together with their constraints. This step is useful for performance as well as random stability reasons.
7. Orders randsets. The order of randsets depends on the nature of random variables or constraints. Generally they are ordered as follows:
    a. Randsets holding cyclic random variables (declared with the *randc* modifier)[3]. Because *randc* cycles operate on single variables, this implies that each *randc* variable must be evaluated separately (even from other dependent *randc* variables).
    b. Randsets holding random variables passed as arguments to functions used in constraints.
    c. Remaining randsets.
8. Picks the appropriate engine for each randset to solve the problem of random variables and constraints at hand.
9. Attempts to solve a randset satisfying all enclosed constraints, taking any number of iterations it needs.
10. Following a randset solution, records the solution within the Solver and then proceeds to the next randset. Take into account that there is no going-back strategy if a subsequent randset fails (i.e. There is no loop back to pick other solutions for previously solved randsets when following randsets fail).
11. If the Solver fails to solve a specific randset, it aborts the entire randomization process and flags a randomization failure (i.e. the *randomize ()* function shall return zero) without updating any of the successfully solved random variables.
12. If all randsets are successfully solved, the Solver will:
    a. Generate random values for any unconstrained random variables remaining.
    b. Updates all random variables with the newly generated solution.
    c. Calls the *post_randomize()* virtual function recursively in a top-down manner.

D. What are the different types of Solvers?

Binary Decision diagram (BDD), Boolean Satisfiability (SAT), Finite Domain (FD), and others are all types of constraint solvers. Each has its own pros and cons; an efficient Solver is usually a hybrid of different engines.

[3] Note that the solution order of different *randc* variables is undefined.

III. UNEXPLAINED RANDOMIZATION FAILURES

Typically, randomization failures occur when constraints contradictions occur. However, often spotting out the constraints contradictions root cause is not a straightforward task. This section highlights the root causes of common unexpected randomization failures gotchas during simulation runtime, and the means to resolve them.

A. My randomization attempt failed and I was not notified

Problem Description

The *randomize()* method by default returns 1 if it has successfully set all the random variables and objects to valid values; otherwise, it returns 0. Also, a randomization attempt is atomic all-or-none procedure. That means for any constraint that cannot be satisfied, *randomize()* does not update any random variable and returns a status of 0. If the return status of *randomize()* was not captured/checked, then you force the simulator to *void* cast the result of *randomize()*. This can result in a silent failure, which could waste your time trying to figure it out.

```
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;};
```

Illustration & Remedy

Always capture the randomization attempt result. There are many ways to do so, one of the best ways is to wrap your randomization with an *if* condition. For example:

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

The second method is to wrap your randomization with an immediate *assert* statement. This is easy to add and makes your code more compact. However, sometimes it could also result in unexpected scenarios; e.g. if you forgot and used *$assertoff()* that was applied to these kind of assertions, or if you switched off immediate assertions firing via your simulator settings[2].

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

This third method is to go for a simulator-dependent way to capture randomization failures. Generally, this is not the recommended method as it is dependent on simulator configurations that could be easily missed, and hence cause unexpected behaviors.

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

Problem Description

Randomization failures occur after constructing your object and randomizing a single variable of it.

```
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
```

Illustration & Remedy

There is a common misconception about single variable randomization, or randomization of a small subset of the entire randomization cluster, that the Solver will focus on randomizing only this variable and ignore everything else. This is NOT true. The Solver will randomize the variable while keeping into consideration all

other constraints in the cluster. In the previous example, one of the constraints denotes that "b" is smaller than "a". Initially, "a" has a value "0", and since "b" cannot hold a negative value (unsigned *bit* data type), the randomization fails. Think of it as if the Solver is generating current values for all random variables not passed to *randomize()*. So, before initially randomizing a single variable, randomize the entire cluster first. This way, the remaining random variables will take values that satisfy all constraints rather than holding initial values. Also note that you can still get randomization failures if your constraints depend on non-random variables that change dynamically in a way not to satisfy any of the constraints enclosed in the randomization cluster.

```
assert (t1.randomize);
assert (t1.randomize (b));
```

C. I am encountering cyclic dependency errors between random variables and constraints

Problem Description

Cyclic (or circular) dependency errors occur when using functions in constraints, or using the *solve..before* constraint.

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

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

Illustration & Remedy

Random variables passed as function arguments are forced to be solved first by the Solver. In the preceding example, the entire "a" vector is solved first by the Solver. The Solver does not look into the details of the functions or how it evaluates its output. Once it picks a random value of "a," it will substitute with this value in the function to get the required value for the LHS, namely, "a[0]". There is a 50% probability that the function would return a value that matches the value the Solver picked for "a[0]". Otherwise, it would be a randomization error. There are many ways to solve this; for instance: 1) only pass "a[7:1]" as a function input argument. 2) Or a much better solution, remove the constraint altogether and use *post_randomize()* to assign "a[0]".

```
constraint c {a[0] == foo (a[7:1]);}
```

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

D. I am encountering cyclic dependency errors between randc variables

Problem Description

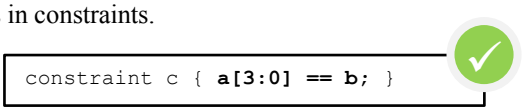
Cyclic (or circular) dependency errors occur when using *randc* variables in constraints.

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

Illustration & Remedy

Variables declared as *randc* are solved before other *rand* variables in the randomization cluster. The SystemVerilog LRM only describes the cyclic nature of the values produced by individual *randc* variables, while it says nothing about any kind of cyclic behavior of solutions from multiple related *randc* variables. Because *randc* cycles operate on single variables, this implies that each *randc* variable must be evaluated separately (even from other dependent *randc* variables). Taking this into consideration, in the example above, if "a" is solved before "b" then value chosen for "a" may not fit into 4 bits to match "b". A randomization failure would occur as it will be impossible for the Solver to satisfy the above constraint. The best remedy is to beware equality between

unmatched size *randc* variables. This theory also holds true for any other explicit solving order inferred from constraints, e.g. using methods in constraints.

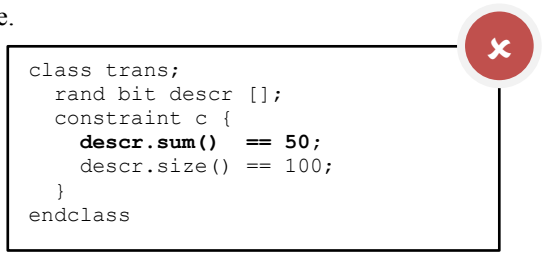


```
constraint c { a[3:0] == b; }
```

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

Problem Description

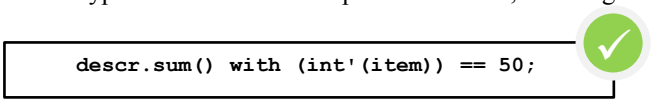
Using *array.sum()*, or *array.product()*, to constrain the summation of all array elements generated values, results in a randomization failure.



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

Illustration & Remedy

Array reduction methods such as *sum()* and *product()* specify that the sum/product is performed using the width of the array elements. If the array in question is an array of bits, the sum/product is computed with a width (precision) of 1-bit. Not taking this into consideration often leads to unexpected results and randomization failures. Take the above example, a correct way to write the above constraint is to explicitly cast the array element (i.e. *item*) to an *int* data type. This ensures the expected behavior, avoiding size reduction and overflow.



```
descr.sum() with (int'(item)) == 50;
```

IV. UNEXPLAINED RANDOMIZATION RESULTS

In real life, figuring the root cause of unexpected random results is not trivial; typically unexpected random results are observed from a testcase failure or a functional mismatch, denoting a long time wasted during debug. This section highlights some of the scenarios that result in unexpected randomization results.

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

Problem Description

Minimal code modifications change the random values generated from run to run, although running with same code revision, simulator revision, seed, simulation commands, and environmental settings.

Illustration & Remedy

Random stability is a major issue when it comes to day-by-day development; we normally seek (and expect) identical generated random values upon minimal code changes. Random stability is crucial in order to: 1) Get consistent results that are important for analysis, development, and verification closure. 2) Replicate bugs, eliminate their escape, and test bug fixes. The element responsible for generating random values in SystemVerilog is called Random Number Generator, abbreviated RNG. Each thread, package, module instance, program instance, interface instance, or class instance has a built-in RNG. Thread, module, program, interface and package RNGs are used to select random values for *$urandom()* (as well as *$urandom_range(), std::randomize(), randsequence, randcase, and shuffle()*), and to initialize the RNGs of child threads or child class instances. A class instance RNG is used exclusively to select the values returned by the class's predefined *randomize()* method[3].

Whenever an RNG is used either for selecting a random value, or for initializing another RNG, it will "change state" so that the next random number it generates is different. Therefore, the random value a specific randomization call generates depends on the number of times the RNG has been used, and on its initialization. Initializing the RNG, in turn, depends on the number of times its parent RNG has been used and on the parent RNG initialization. The top most RNG is always a module, program, interface or package RNG, and all of these RNGs are initialized to the same value, which is chosen by the simulator according to the simulation seed. For a

given *randomize()* call, the process is essentially the same up to the point where the object is allocated. Once the object is allocated, it gets its own RNG which, unlike package, module, program, interface or thread RNGs, changes state only when *randomize()* is called. Therefore, from instantiation point onwards, the only instructions that affect the results of a given *randomize()* call are earlier randomize() calls. In the example below, the random scenarios generated by randomizing the "rw_s" sequence will totally change when instantiating a new sequence object "rand_s" before the "rw_s" sequence object instantiation.

```
virtual task body;
  random_seq rand_s = new; //Affects random stability of Line A
  simple_seq rw_s   = new;
  fork begin
      assert (rw_s.randomize()); //Line A: Randomize "rw_s" test sequence
      rw_s.start(); //Drive the sequence
    end
    ...
  join
endtask
```

Instead of depending on the absolute execution path for a thread, or on the ordering of an object construction, the RNG of a given thread or an object can be manually set to a specific known state. This makes the execution path up to a point "don't care". This is known as *manual seeding*, which is a powerful tool to guarantee random stability upon minimal code changes. Manual seeding can be performed using:

- srandom(): Takes an integer argument acting as the seed. Once called on a process id or a class object, it manually sets the process (or object) RNG to a specific known state, making any subsequent random results depend only on the relative execution path from the manual seeding point onwards.
- get_randstate()/set_randstate(): Used together to shield some code from subsequent randomization operations.

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

Random stability is addressed carefully in the Universal Verification Methodology (UVM)[4]: a) UVM components are re-seeded during their construction based on their type and full path names. b) Sequences are re-seeded automatically before their actual start.

B. Unexpected negative values are generated upon randomize

Problem Description - 1

Randomization attempts do what you ask them to do. If you gave them signed types, their solution space will accommodate for negative values as well. This rule applies to any variable declared as *signed,* as well as variables of type *int* or *byte*. Not taking this into consideration can result in performance penalty, unexpected results, and sometimes randomization failures.

Illustration & Remedy - 1

Always check the sign nature of your random variables and make sure you are not mistakenly defining variables as *signed*, and vice versa. I.e. 1) Do not use the *signed* modifier when not needed, 2) For 7-bit variables of unsigned nature, use "*bit [7:0]*" instead of *byte* data type, 3) For 32-bit variables of unsigned nature, use "*bit [31:0]*" instead of *int* data type.

Problem Description - 2

Issues may also arise the other way around, that is, when using unsigned data types to hold negative values. Take a look at the following example:

```
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]]});
    }
  }
  length == 6'h10;
}
...
start_end_addr_hash [start_addr] = start_addr + length - 1;
```

Illustration & Remedy - 2

The constraint "c" above is intended to eliminate overlapping of address ranges between successive randomization attempts. The only exception of the above constraint is when "length" is greater than "i+1" (e.g. Imagine the case where a "start_addr" of a previous randomization attempt was picked to be smaller than the address range "length"). In this case, the constraint will always hold TRUE. The reason is that negative numbers are represented as 2's complement when assigned to unsigned data types (like "start_addr" above). So in the case demonstrated above, the constraint would be of the form "`!(start_addr inside {[<HIGH-VALUE> : <LOW-VALUE>]});`" which always holds to TRUE and hence the Solver could generate overlapping address ranges below the "length" value. To prevent this type of error occurrence, you need to anticipate these corner cases and add guard expressions to avoid them.

```
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]]});
  }
}
```

C. My rand dynamic array is not constructed after randomization

Problem Description

Dynamic arrays can be declared as *rand*. Take the following example:

```
class c;
  rand bit [7:0] dyn_arr[];
endclass
c c1 = new;
assert(c1.randomize());
```

You would expect that after the *randomize()* call the dynamic array will be generated with an arbitrary size and its elements will be randomized; however, this is not the case! The randomize call here will exit with no error or warning, and the dynamic array will not be resized, retaining its previous size, 0.

Illustration & Remedy

The SystemVerilog LRM states that the size of a dynamic array or queue declared as *rand* or *randc* can also be constrained. In that case, the array shall be resized according to the size constraint. If a dynamic array's size is not constrained, then the array shall not be resized. Initially the size of the dynamic array is zero.

```
assert(c1.randomize() with {dyn_arr.size() < 10;});
```

Another important aspect is that the Solver will NOT instantiate new class objects when resizing a dynamic array of class handles, this sometimes result in unexpected runtime fatal errors especially to people with other HVLs background. It has to be carefully kept in mind that *randomize()* does not instantiate class objects.

D. Output random distribution is not expected when using the "dist" operator

Problem Description

When using *dist* constraints on the following form, the Solver always generates some values ("12" and "31") and never generates others.

```
constraint c { x dist {[0:10]: 1/11, 12: 1, [13:30]: 1/18, 31}; }
```

Illustration & Remedy

The intention of the above constraint will not be fulfilled by the user coding. The intention of the constraint is to specify a probability of "1" to be shared across elements "[0:10]" (11 elements) and "[13:30]" (18 elements). However, what happens is that due to integer division operation, results are truncated to 0. Hence, the Solver would only generate values "12" and "31". The correct coding to implement the intended constraint is already supported by the SystemVerilog language as follows:

```
constraint c { x dist {[0:10] \:1, 12:1, [13:30]\:1, 31}; }
```

E. Output random distribution is not perfectly cyclic although I am using randc

Problem Description

Even though I defined some variables as *randc*, the generated random results are not perfectly cyclic. Take the following example:

```
class c;
  randc bit [1:0]   a ;
  randc bit [4:0]   b ;
  constraint c_trans {
    (a != 2'b01) -> (b <  5'h10); //#1
    (a == 2'b01) -> (b >= 5'h10); //#2
  }
endclass
```

Illustration

Even when variables are explicitly defined with the *randc* modifier, their intended cyclic random behavior can be compromised upon constraints dependencies. The order in which *randc* variables are solved is tool dependent as not defined by the SystemVerilog LRM, so if the tool chooses to solve one of the variables first, it can compromise the cyclic nature of the second variable. So the Solver here has two options: either to throw a randomization failure, or to compromise the intended cyclic random behavior of one of the *randc* variables for the randomization attempt to be successful.

F. My inline constraints are not applied

Problem Description

The following example shows an attempt to randomize a transaction that constrains the transaction address to be equal to the calling sequence address. However, "t.addr" and "seq.addr" are not equal after the randomization attempt!

```
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
```

Illustration & Remedy

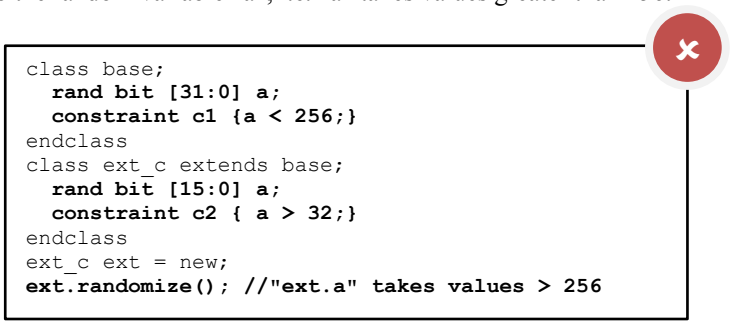
The SystemVerilog P1800-2012 LRM states, "*Unqualified names in an unrestricted in-lined constraint block are then resolved by searching first in the scope of the randomize() **with** object class followed by a search of the scope containing the method call—the local scope*". So the above constraint was actually constraining "t.addr" to be equal to itself. The *local::* qualifier modifies the resolution search order. When applied to an identifier within an in-line constraint, the *local::* qualifier bypasses the scope of the [*randomize() with* object] class and resolves the identifier in the local scope. The correct coding of the above example would be:

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

G. Base class constraints are not applied to the extended class

Problem Description

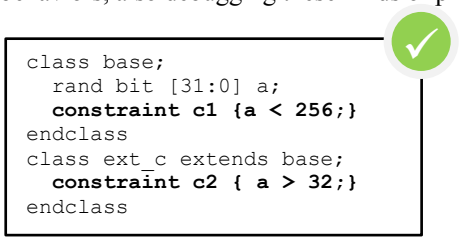
The following example shows an attempt to randomize an object of class "ext_c." The constraint "c1" in "base" is not applied to the random variable "a", i.e. "a" takes values greater than 256.



```
class base;
  rand bit [31:0] a;
  constraint c1 {a < 256;}
endclass
class ext_c extends base;
  rand bit [15:0] a;
  constraint c2 { a > 32;}
endclass
ext_c ext = new;
ext.randomize(); //"ext.a" takes values > 256
```

Illustration & Remedy

The "ext_c" class defines a variable with the same name "a" as the "base" class. This is perfectly legal in SystemVerilog, each variable will have its own context in objects of type "ext_c" (i.e. "ext.a" will access "a" defined in "ext_c", while "ext.super.a" will access "a" defined in "base"). The constraint "c1" in this case will always be applied to "a" defined in "base", and will not be applied to "a" defined in "ext_c". It is usually not a good idea to define variables with the same name in extended and base classes; it makes your code prone to many runtime errors and/or unexpected behaviors, also debugging these kinds of problems may not be trivial.



```
class base;
  rand bit [31:0] a;
  constraint c1 {a < 256;}
endclass
class ext_c extends base;
  constraint c2 { a > 32;}
endclass
```

Other scenarios with the same symptom (a constraint being ignored), could occur when:

- Constraint c1 is a *soft* constraint that is contradicted/overridden in an extended class or inline constraint.
- Calling *this.randomize()* from the "base" class *new()* method. Since *randomize()* is a virtual function, one may expect that constraints defined in the extended class would take effect in this case however this may not necessarily be the case. When calling virtual methods in constructor, extended class properties are not allocated yet. The behavior of a virtual method call in a constructor is undefined in the LRM, and hence is tool implementation dependent. As a remedy, avoid calling *randomize()* in classes' constructors.

H. Random values generated change from run to run when running with different simulators.

Problem Description

The simulation runtime random variables generated are different when running on different simulators, although running the same source code revision with the same seed, environment, and equivalent commands.

Illustration & Remedy

Different simulators use different constraint solvers that cannot be compared to each other; a simulator A invoked with initial seed S, would probably generate totally different random stimulus than simulator B invoked with the same initial seed S. This might even be true for different versions of the same simulator. So if you tend to use different simulators in your daily verification tasks make sure to: 1) Build reference models and self-checking testbenches so that different constrained random values generated during simulation may not be troublesome. 2) Build Coverage models to assess tests' effectiveness. 3) Stick to the same version of the same simulator and the same seed during debug times or when reproducing failures. 4) Leverage manual seeding for random stability.

I. I am getting unexpected random results when using default constraints

Problem Description

Although a default constraint is defined, random values generated are not compliant with the constraint. Take a look at the below example, sometimes values generated for "y" are smaller than "z".

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

Illustration & Remedy

Default constraints are not part with the SystemVerilog P1800-2012 LRM; they come from the *Vera* language. Several simulators allow them as a sort of an extension to the SystemVerilog language. Default constraints can be specified by placing the keyword *default* ahead of a constraint block definition. They are constraints acting as soft contracts between the user and the Solver. However, once any variable used in them is used in another constraint (it does not matter if this constraint contradicts the default constraint or not), the entire default constraint is ignored. In the example above, the constraint "c2" does not contradict the default constraint "c1". However, the entire default constraint "c1" is ignored by the Solver since the variable "x" that appears in constraint "c1" appears in constraint "c2" as well. This can be a serious problem, since the constraint of "y > z" will be ignored too, although no other constraints access "y" or "z" random variables.

As a rule of thumb, do NOT use default constraints. They provide no additional value over what is already defined in the SystemVerilog language. On the other hand, they could cause unexpected results. Instead, use *soft* constraints or enable/disable constraints via the *constraint_mode()* method.

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

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

Problem Description

Imagine a design that contains some C code that performs some random generation while the C implementation is connected to the SystemVerilog implementation using the Direct Programming Interface (DPI-C). However, the initial simulation seed is not affecting random numbers generated by the C code.

Illustration & Remedy

Normally, the initial SystemVerilog simulation seed, set by the simulator or by the user via simulation *plusargs*, affects the SystemVerilog code only; it does not affect the foreign language code. This can be resolved by passing the simulation initial seed to the foreign language (e.g. C/C++) code as follows:

1. Import in SystemVerilog code a C function that takes a seed as an argument.
2. Call the C function in the SystemVerilog code at the beginning of simulation passing the initial seed (or a random value seeded by the initial seed).
3. The C function will call *srand()* passing the initial seed (or a random value seeded by the initial seed).

```
// 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;
}
```

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

REFERENCES

[1] IEEE Standard for SystemVerilog, *Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2012, 2012.

[2] Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them, Stuart Sutherland and Don Mills, Springer.

[3] UVM Random Stability: Don't leave it to chance, Avidan Efody, DVCon 2012.

[4] UVM User Manual, uvmworld.org.

[5] Verification Academy, www.verificationacademy.com.