Randomizing UVM Config DB Parameters

Jeremy Ridgeway
Avago Technologies, Inc.
Fort Collins, CO 80525
jeremy.ridgeway@avagotech.com

***Abstract*-The Universal Verification Methodology (UVM) library provides two significant benefits to simulation configuration. It has a flexible configuration database that is globally accessible and parameters may be set via the command-line without simulation recompilation. In this paper we describe how we have enhanced the UVM configuration database with a random-layer and command-line access by connecting a limited set of random constraints. The end result is that random constraints may be applied to UVM configuration database parameters directly on the command-line. Code accessing the database need not specify random variables and constraints to benefit from randomness.**

I. INTRODUCTION

Verification environment configuration with the Universal Verification Methodology (UVM) library provides two significant benefits. First, the configuration database is flexible and globally accessible to the environment as a whole via a static class dereference operation, `uvm_config_db#(T)::get()`. Second, the parameter values stored in the configuration database may be set on the command-line without simulation recompilation [1, 2, 3].

We have found that often configuration parameters that are initially set directly will later in the project be required to change to random variables. Of course, SystemVerilog random constraints do not play well with UVM configuration; the actions tend to differ. A workaround could involve a second configuration parameter indicating if the first should be randomized or take its value directly from the configuration database. In fact, this is precisely what we had in mind when developing direct randomization on scalar UVM configuration database parameter access. However, instead of two separate parameters (a control flag and a data value), we combined them into one.

We had two goals to accomplish with this technique. First, we had to enable randomization on existing scalar type configuration parameters with little to no change to source code. Second, the technique had to support indicating random constraints on the command-line. We achieved both, in a limited set, by building on the dynamically interchangeable constraints described in [4] and introducing a new random "convenience" layer to the UVM configuration database.

Already, we had a library of type-parameterized random variable container classes capable of instantiating constraints specified in a string. The SystemVerilog constraint, then, is created on-the-fly and affects the value in the random variable container at randomization time. We tied this container class to a resource database convenience layer extended from the `uvm_config_db` interface class, named `config_rand_db`. This new layer opened the run time configuration option to become randomizable via a constraint specified on the command-line, even though the option was *not* declared **`rand`**. Then, a simple code changed was reuqired in the verification environment where the UVM configuration database is accessed. For example:

```
int num_actors;
uvm_config_db#(int)::get(this, “”, “num_actors”, num_actors);
```

We changed the above UVM configuration database access to access the random convience layer instead.

```
int num_actors;
config_rand_db#(int)::get(this, “”, “num_actors”, num_actors);
```

Now, we were able to randomize configuration options on the command-line. For example:

```
> simcmd +uvm_set_config_string='*.bus_env,num_actors,inside [1:2]'
```

This paper is organized as follows: first we review interchangeable random constraints from [4] in section II. Next we recap the available UVM command-line access approach in section III. Then we present our random-specific convenience layer to the UVM configuration database and its automatic command-line access in section IV. Finally, we conclude in section V.

II. Interchangeable Random Constraints

In [4], we defined a set of random variable container classes to implement a subset of the SystemVerilog constraint language. When these containers are coupled with a front-end parser and test bench-wide resource manager, such as the UVM configuration and/or resource database, constraints may be fully interchanged on-the-fly. Furthermore, neither deep test bench architecture nor verification methodology need be known to affect and change constraints. In this section we review the container class library as well as its front-end parser, as described in [4].

A. SystemVerilog Constraints

Generally, SystemVerilog random variables and their constraints exist in the same or inherited class. Furthermore, the constraint itself is largely declarative in the SystemVerilog language. That is, the constraint must be fully resolved at compile- and elaboration-time, prior to simulation. Some flexibility exists by using class variables to shape the randomization (e.g., a minimum and maximum such that min ≤ val ≤ max). Through inheritance, a named constraint block may be overridden with a new constraint or removed by setting to an empty block. Constrained ranges may be further restricted through inheritance by adding new constraint blocks. In all cases, a fair amount of knowledge of the class and its use in the verification environment is required.

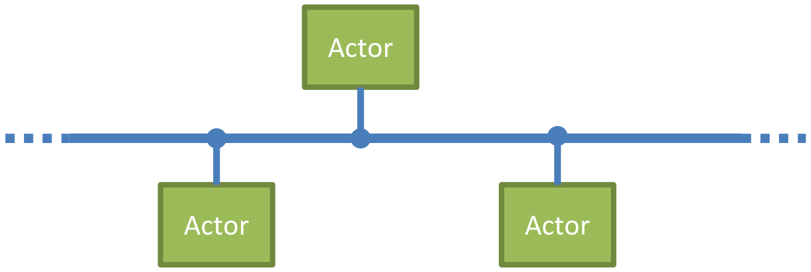

Figure 1: Example bus architecture under test. Actors are verification components whose number is chosen at run-time.

Consider, as an example, some bus architecture under test, refer to Figure 1. Suppose there can be a fixed number of actors attached to the bus as determined by a simulation time configuration option. An “actor” is encapsulated as some in-house or third party verification intellectual property (VIP) UVM agent class.

```
class bus_env extends uvm_env;
    bus_actor_agent actors[]; // Number determined at sim time
    int num_actors;           // Valid range is 1-10

    function void build_phase(uvm_phase phase);
        if(!uvm_config_db#(int)::get(this, “”, “num_actors”,
                                     num_actors)) // Store locally
            `uvm_fatal(“CFG_ERR”, “Number of actors unknown”)
        …
    endfunction
endclass
```

The number of actors, in the code above, must be known to the bus environment to build properly; otherwise a fatal error will exit the simulation. An alternative to the above is to use a second configuration option to indicate the value should either be taken from the configuration database or independently randomized, as in the code below.

```
class bus_env extends uvm_env;
    bus_actor_agent actors[]; // Number determined at sim time
    rand int num_actors;      // Valid range is 1-10
    constraint num_actors_c { num_actors inside {[1:10]}; }

    function void build_phase(uvm_phase phase);
        int randomize_num_actors; // Control flag: config_db or rand
        if(!uvm_config_db#(int)::get(this, “”, “randomize_num_actors”,
                                     randomize_num_actors) begin
            if(!uvm_config_db#(int)::get(this, “”, “num_actors”,
                                         num_actors)) // Store locally
                `uvm_fatal(“CFG_ERR”, “Number of actors unknown”)
        end else begin
            this.randomize(num_actors); // Instructed to randomize
        end
    endfunction
endclass
```

The above code allows the number of actors to be set via the UVM configuration database *or* randomized by the valid constraint. When the user indicates the `num_actors` option should be randomized, then the `num_actors_c` constraint applies. Changing the SystemVerilog constraint requires an extension to `bus_env` in some test code, then registering with the factory an override using the new `bus_env` in place of the old. Still the `randomize_num_actors` configuration option must be set in order for the new constraint to take effect.

```
class bus_env_limited extends bus_env;
    constraint num_actors_c { // Limit to 1-3
        num_actors dist {1:=15, 2:=50, 3:=35};
    }
endclass

class test_bus_env_limited extends base_test;      // UVM test type
    function void build_phase(uvm_phase phase);
        uvm_factory factory uvm_factory::get();
        factory.set_inst_override_by_type(bus_env, // orig
                                          bus_env_limited, // new
                                          “*”);    // everywhere
    endfunction
endclass
```

To achieve SystemVerilog constraint overriding, both the leaf class name containing the random variable and the constraint block name affecting the random variable must be known. Then, the new class extension must override the original class via the UVM factory. This flexible feature of UVM requires a fair amount of knowledge of the environment as well as the UVM library.

B. Random Variable and Constraint Container Classes

In [4], we defined a set of random variable containers, the boxes labeled `loc_rand` in Figure 2, with separate constraint containers, the box labeled `constrain`. This methodology capitalized on somewhat of a loophole in SystemVerilog constraints: a constraint may be applied over class references and that referencing need not be defined immediately. This allowed the constraint itself to be instantiated in a class separate from, but act upon, the random variable container.

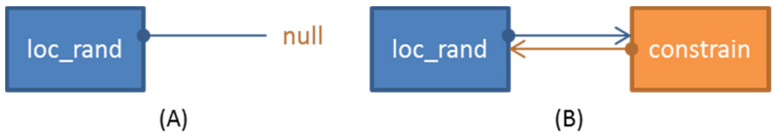

Figure 2: Random variables with dynamic constraints: (A) unconstrained, (B) constrained.

The `loc_rand`[1] class is broken into an inheritance tree of four classes, refer to Figure 3. The single data member maintained by the classes, **`rand`** `T value`, where `T` is a parameterized type indicated at declaration time, is a member of the `loc_static_rand` base class. The name implies static constraint access. In other words, the constraint on the random value is *not* updated on-the-fly at randomization time. The `loc_param_rand` base class maintains a constraint class reference member, `T constrain`, as well as a reference to the type-specific static constraint factory (see section II-C). When the reference points to **`null`**, as in Figure 2-A, the value member is randomized *unconstrained*. However, when the reference points to a constraint container class instance, as in Figure 2-B, then the value member is randomized according to the specified constraint. Finally, the `loc_rand` class maintains the capability to dynamically update the constraint at randomization time.

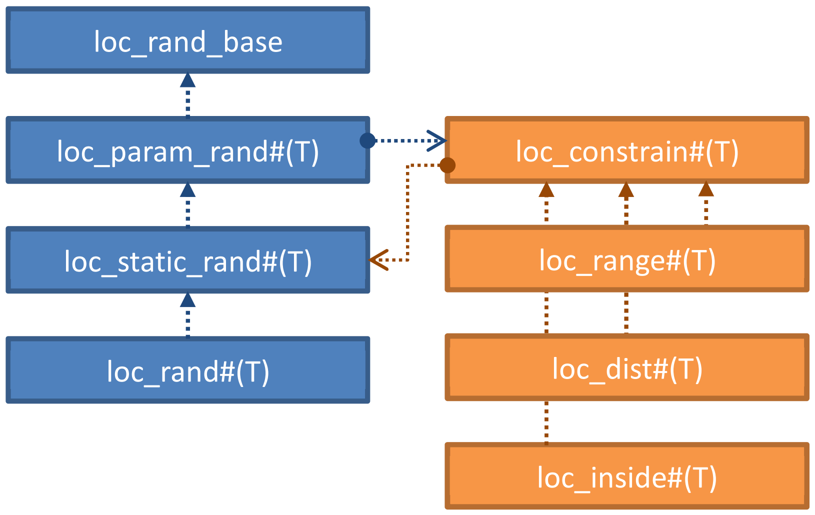

Figure 3: Random variable container (left) and constraint container (right) template class inheritance tress. The random variable class refers to an instance of the constraint class, and vice versa.

The constraint is a general purpose container class with a type parameter that agrees with `loc_rand`. We have defined a limited library of constraints applicable to the random variable container class, as in Figure 3. Each specialized container class implements the constraint indicated using local variables as necessary. The actual randomization function is encapsulated in the specialized `next()` function, an abstract function in the constraint container base class.

```
class loc_constrain#(type T = int);
    loc_static_rand#(T) var;
    pure virtual function T next();
endclass
```

The range constraint class implements a uniformly random contiguous range. Two local access class members are set at constraint instantiation (as specified in the string). The random variable container class is randomized via a **`randomize`**`()`**`with`** construct.

[1] In [4], `loc_rand` is named `lvm_rand`.

```
class loc_range#(type T = int) extends loc_constrain#(T);
    local const T min;
    local const T max;
    virtual function T next();
        var.randomize() with {
            value inside { [local::min : local::max] }; };
        return var.value;
    endfunction
endclass
```

The inside set constraint class implements a uniformly random selection of a distinct set of values. A single local access queue is populated at construction with values from the constraint (as specified in the string). The random variable container class is also randomized via a **randomize()with** construct.

```
class loc_inside#(type T = int) extends loc_constrain#(T);
    local T setlist[$];
    virtual function T next();
        var.randomize() with {
            value inside { local::setlist };
        };
        return var.value;
    endfunction
endclass
```

The distribution constraint class implements a random selection of a distinct set of values based on specified weights. A pair of queues is populated at construction with values and weights as positive and non-zero integer numbers (as specified in the string). This randomization occurs in two steps. First, a weight is randomly selected from the total weight range, in the `next_sel()` function. Then, the variable container class is randomized via a **randomize()with** construct.

```
class loc_dist#(type T = int) extends loc_constrain#(T);
    local T setlist[$]; // Set of values to choose from
    local int weightlist[$]; // Weight for each value
    local rand int weight_sel; // Current weight selection
    local int listsel;         // Current list selection

    function void next_sel();
        int mn = 0, mx = 0, tot = 0;
        foreach(weightlist[i]) // Accumulate all weights
            tot += weightlist[i]);

        this.randomize(weight_sel) with { weight_sel inside [1:tot]; };

        foreach(weightlist[i]) // Find the index for selected weight
            if(weightlist[i] > 0) begin
                mx += weightlist[i]; // Set upper bound
                if(weight_sel > mn && weight_sel < mx) begin
                    listsel = i; // index chosen based on random weight
                    break;
                end
                mn = mx; // Set lower bound for next loop
            end
    endfunction

    virtual function T next();
        next_sel();
```

```
            var.randomize() with { value == local::setlist[listsel] };
            return var.value;
        endfunction
    endclass
```

The random variable container class, `loc_rand_static`, also implements the specialized `next()` function to retrieve a new random value.

```
    class loc_rand_static#(type T = int) extends loc_rand_param#(T);
        rand T value;
        virtual function T next();
            if(m_constraint == null) begin
                this.randomize(value);
                return value;
            end else begin
                return m_constraint.next()
            end
        endfunction
    endclass
```

Together, the random variable container class and the constraint container class generate type-specific random values. Coupled with a parser front-end, the constraint reference may be instantiated on-the-fly, deleted, and instantiated again.

C. Dynamic Constraint Creation

The constraint for the random variable may be instantiated by the user directly, but dynamism is achieved when coupled with a parser and a constraint factory (similar to the UVM factory for objects), as in Figure 4.

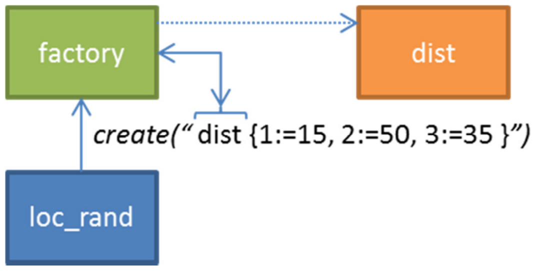

Figure 4: Factory parses the string and instantiates the constraint.

The constraint factory is maintained per type in the simulation. The `loc_param_rand` base class maintains a reference to a type-specific constraint factory.

```
    class loc_rand_base;
        // Empty class for non-parameterized referencing
    endclass

    class loc_param_rand#(type T = int) extends loc_rand_base;
        static constraint_factory#(T) factory;
    endclass
```

The `loc_static_rand` class implements a `push()` function that actually invokes the constraint factory and its parser to instantiate the new constraint from a string.

```
    class loc_static_rand#(type T = int) extends loc_param_rand#(T);
        void function push(string constraint);
```

```
            pop(); // Remove the old constraint
            factory = constraint_factory#(T)::get();
            m_constraint = factory.create(constraint, this);
        endfunction
    endclass
```

Finally, the `loc_rand` class implements dynamic constraint instantiation. The class accesses the UVM configuration database with a UVM component parent context and the local class name. If a string exists at the specified location, then a new constraint is instantiated, replacing any existing constraint. The configuration database is cleared for that constraint to ensure a persistent constraint until the user wishes to change again.

```
    class loc_rand#(type T = int) extends loc_static_rand#(T);
        function new(uvm_component parent, string name); … endfunction

        function void update_constraints();
            string constraint;
            if(uvm_config_db#(string)::get(get_parent(), “”, get_name(),
                                               constraint)) begin
                if(constraint != “”) begin
                    factory = constraint_factory#(T)::get();
                    m_constraint = factory.parse(constraint);
                    uvm_config_db#(string)::set(get_parent(), “”,
                                                     get_name(), “”); // Clear
                end
            end
        endfunction

        virtual function T next();
            update_constraints();
            return super.next();
        endfunction
    endclass
```

The constraint parser takes a string conforming to the grammar specified in [4], and included in the appendix, and builds an instance of the specified constraint type. For example, consider the constraint from Figure 4 and shown Figure 5-A. As the parser traverses the string, a distribution constraint is created. The orange box labeled `dist`, in Figure 5-B, is an instance of the `loc_dist` constraint class. Each purple block indicates a member of the `setlist` queue within the class; while each blue block indicates a member of the `weightlist`. The location of each value in both lists indicates the correlation (i.e., setlist[0] correlates to weightlist[0]).

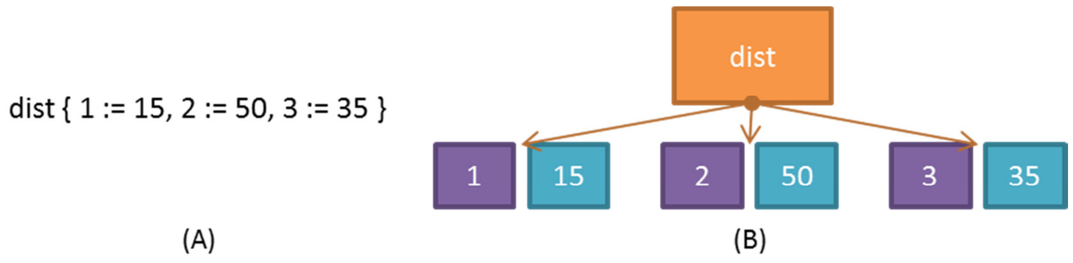

Figure 5: Constraint string (A) and its representation (B).

The constraint factory incorporates a grammar and parser implementing a top-down LL(1) parse. The input string is read from the Left with a Left-most derivation and requiring only 1 look-ahead token to determine the next action. The constraint is created top-down. For our parser, the constraint type is always indicated first (e.g., `dist`), followed by its details. In this manner, the constraint factory works directly with the parser to instantiate the constraint.

Some important items are of note in the methodology:

1. The random variable container, the constraint container, and the constraint factory types all agree,
2. The type is known at compile time,
3. A constraint string is parsed only when required.

The type is also necessary to cast string values appropriately. For example, numbers are identified by the parser as conforming to SystemVerilog syntax and indicating base (e.g., `'h` for hexadecimal). The actual conversion, then, occurs via the SystemVerilog system function `$sscanf`, as in the pseudo-code below.

```
function T get_val(string input_val);
    T arg;
    if(is_hex(input_val))
        if(!$sscanf("32'h47", "%h", arg))
            `uvm_fatal("RAND", "Type conversion failure")
    … // other bases
    return arg;
endfunction
```

The advantage with the built-in system function, `$sscanf`, is proper handling of 4-state data types. For example, a random variable class may be declared as `loc_rand#(logic)`. In this case, the values specified in the constraint string may include Xs and/or Zs. Both the `$sscanf` function and randomize functions handle them correctly. The user is required to determine which constraints are applicable when using Xs or Zs.

Returning to the example `bus_env` in this section, we can use the `loc_rand` class to encapsulate both the configuration option and the configuration database access.

```
class bus_env extends uvm_env;
    bus_actor_agent actors[];   // Number determined at sim-time
    loc_rand#(int) num_actors;  // Valid range is 1-10

    function new(string name, uvm_component parent);
        num_actors = new(this, "num_actors");
    endfunction

    function void build_phase(uvm_phase phase);
        num_actors.next(); // Get constraint from UVM config db
    endfunction
endclass
```

The `loc_rand` accesses the UVM configuration database and command-line strings can be inserted into that database. Therefore, we can control the number of actors with a single `loc_rand` instance. The caveat is that the configuration option must have already been declared as `loc_rand`. That may not always be desirable. Note that the `loc_rand` class must be instantiated before it is used. This wasn't the case when `num_actors` was simply a local class member, and changing requires modification/addition of multiple lines of code. In the next sections we will present how to enable randomness on any scalar SystemVerilog configuration option with a simple (read: scriptable) change.

III. UVM Command-line Access

UVM has two command-line options that seed the configuration database at simulation time:

1. `+uvm_set_config_int`, and

2. `+uvm_set_config_string`.

Both options take three arguments separated by commas: (hierarchical) component name, field name, and value. At time-zero, the static `uvm_root` top component parses all instances of the above options and populates the configuration database. For #1, UVM attempts to convert the value (a string on the command-line in binary, octal, decimal, or hexadecimal) to the 2-state `uvm_bitstream_t` typed number before setting into the configuration database.

For example, return to the example in section II, a verification component `bus_env` with a configurable number of bus actors.

```
int num_actors = 1;
uvm_config_db#(int)::get(this, "", "num_actors", num_actors);
```

A regression script can enable set the actors to 2 via:

```
> simcmd +uvm_set_config_string='*.bus_env,num_actors,2'
```

To select specific configurations during regression, we could create separate scripts *per* configuration option. Or, we would create configuration classes with sets of random constraints and always randomize [5]. Neither option fulfilled our requirement of being able to easily select a specific configuration *and* support random configurations using the existing verification environment.

IV. COMMAND-LINE RANDOM CONSTRAINTS

With the random constraint containers, described in section II, applying randomization to the command-line becomes simple. The command-line is accessed as a string, as is the dynamic constraint instantiation. The crux, then, is knowing the proper type. The UVM configuration database itself solves the type issue.

The UVM configuration database may be accessed in two ways: by string name, and by type. Therefore, the data type to randomize is known directly from the configuration database `get()` function. We add a *random layer* on top of the configuration database with its own `get()` function.

```
class config_rand_db#(type T=int) extends uvm_config_db#(T);
    static function bit get(input uvm_component cntxt,
                            input string inst_name,
                            input string field_name,
                            inout T value);
endclass
```

When the user retrieves a random value by data type, refer to Figure 6, we perform several steps. First, randomization on a configuration database value is considered a *one-time* action. Therefore, the parameter is a candidate for randomization if and only if it has not yet been randomized. Second, a string type value is retrieved from the configuration database for the same database value. If a string is available and is recognized as a constraint string, then a new `loc_rand#(T)` is instantiated, randomized, and value put in place at the configuration database context by instance plus field name. Finally, `uvm_config_db#(T)::get()` function is called normally. The remainder of this section presents details for each step.

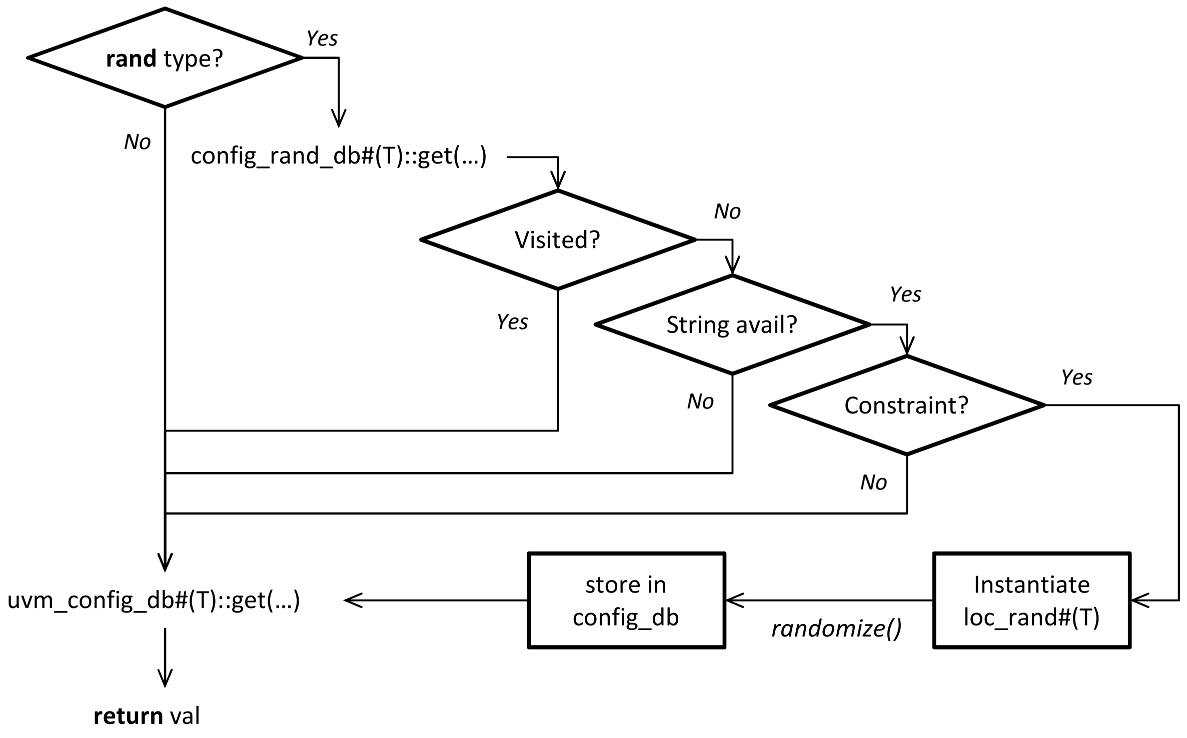

Figure 6: Randomizing a configuration database parameter.

D. *Randomize a Subset of Types*

Our random convenience layer to the configuration database may only be utilized if the data type *can* be randomized. Currently, randomization applies to a subset of SystemVerilog data types [6]. These are limited to the built-in 2-state and 4-state scalar variable types, signed or unsigned, and user defined types that resolve to these (e.g. enumerations). A complete list of data type candidates for randomization is:

- 2-state: shortint, byte, int, longint, all signed or unsigned;
- 2-state: bit or variable length bit-vector;
- 4-state: integer;
- 4-state: logic or variable length logic-vector.

Real numbers and strings are not candidates for randomization and should not use the random-layer access. Furthermore, non-scalar types, including aggregates (queues or arrays), are not supported by the random container class described in section II. Individual members of complex data types or aggregates may be populated, however, by the configuration database random-layer functions.

E. *One-time Randomization*

The goal for configuration database parameter randomization is to easily enable random constraints directly from the command-line on otherwise non-random variables. As such, its use model targets a one-time randomization. A static hash table is employed in the random-layer's static get function to ensure the contextual value moves through the flow exactly once, refer to Figure 6. Subsequent accesses only retrieve the previously randomized value from the configuration database.

```
class config_rand_db#(type T=int) extends uvm_config_db#(T);
    static bit visited[string];
endclass
```

The `visited` hash, as in the code above, is implemented as an associative array with a string key mapping to a Boolean value. The key is composed of the full string path in the `get()` function call.

```
string key = (cntxt != null && inst_name != “”) ?
                 { cntxt.get_full_name(), “.”,
                   inst_name, “.”, field_name } :
             (inst_name != “”) ?
                 { inst_name, “.”, field_name } :
             (cntxt != null) ?
                 { cntxt.get_full_name(), “.”, field_name } :
             field_name;
```

If the key already exists in the `visited` hash table then only the normal configuration database get action is performed.

F. Retrieving the constraint

For configuration database parameters that are candidates for randomization (can be randomized, not yet visited), a string constraint is first retrieved. There are two approaches to string retrieval. The configuration database may be used exclusively to locate the string type. In this case, the original `config_rand_db#(T)::get()` function call is translated to a `config_rand_db#(`**`string`**`)::get()` function call. If the string conforms to the constraint grammar for the random container class, refer to section II, then it is used for randomizing the configuration database parameter. In this case, the UVM built-in command-line argument is employed to populate the configuration database:

```
+uvm_set_config_string=<comp>,<field>,<value>
```

While the value argument may be numeric, the UVM library interprets it as a string stored at the location(s) specified by component plus-arg field names. Wild cards may be employed for a part or the whole component name to apply to multiple database locations.

As an alternative approach, the UVM command-line processor may be used to populate the configuration database when necessary. This action, as described in the modified flow in Figure 7, provides three specific advantages: (a) lazy access, (b) direct command-line plus-arg addressing, and (c) improved SystemVerilog number handling.

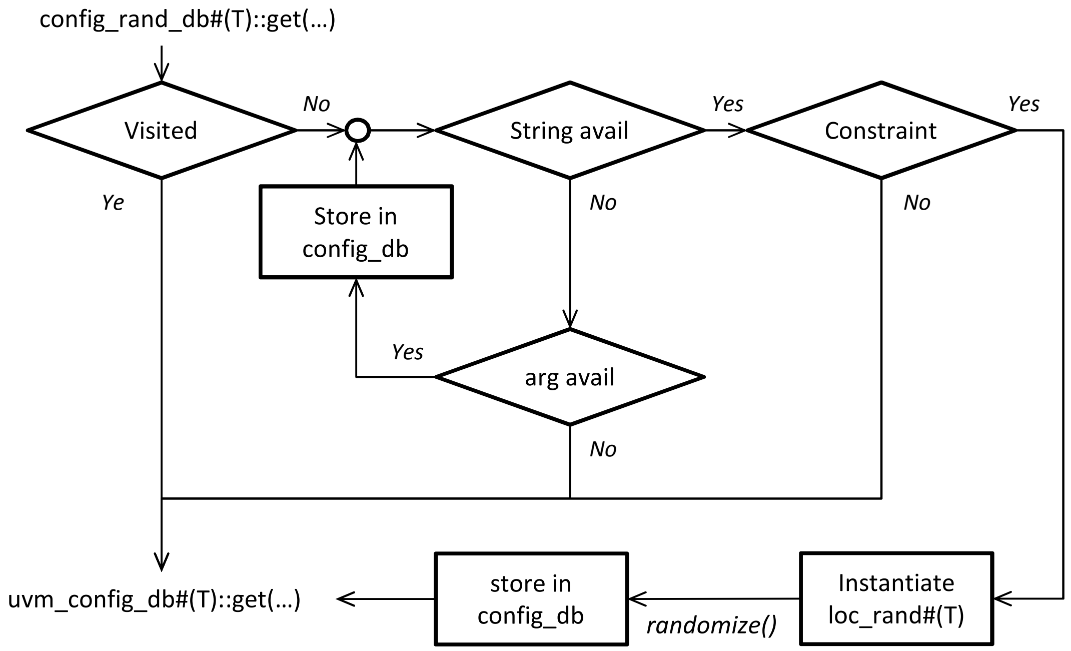

Figure 7: Override UVM configuration database command-line access.

The UVM library populates the configuration database immediately at simulation start based on the command-line plus-args, or command-line arguments that begin with the plus sign. With the lazy approach, however, command-line plus-args do *not* exist in the configuration database and therefore do not affect search until they are required.

While the built-in UVM configuration database plus-arg options have flexibility, our team found addressing the parameter directly as its own plus-arg to be more natural. Our approach uses the UVM command-line processor to retrieve arguments in one of the two following forms:

- `+<context>.<field>=<value>`, or
- `+<field>=<value>`.

Wild-cards are implied for the left-most component name. Thus, `+<field>` implies:

```
+uvm_set_config_string="*,field,value"
```

Finally, the UVM library currently supports simple C-style numeral formatting, excluding Xs and Zs. We enhance this by supporting both the C-style as well as the full SystemVerilog format, including Xs and Zs.

G. *Randomizing the parameter*

Once a string is identified for the configuration database parameter, the typed random container class is instantiated and value randomized. At this point all the following are known:

- Type – scalar data type to randomize,
- Constraint – string to apply to randomization,
- Context – location in configuration database.

Now, a random container class may be instantiated with appropriate type.

```
// Use the key defined earlier:
//   context, instance name, field name, as appropriate
```

```
loc_rand#(T) lrnd = new(key, uvm_top);
lrnd.push(constraint_string);
```

The new constraint, retrieved from the configuration database, is pushed onto the `lrnd` instance. The `loc_rand` type is the same type specified in the `config_rand_db#(T)::get()` function call. Then, a parser works with a constraint factory to parse the string and instantiate the appropriate constraint class, refer to Figure 2 and Figure 4. If this action fails, then the random-layer aborts randomization and just performs the normal configuration database `get()` access. Otherwise, `lrnd` is randomized via the `next()` function call and the resulting value is set into the configuration database.

```
uvm_root uvm_top = uvm_root::get();
uvm_config_db#(T)::set(uvm_top, inst_name, field_name, lrnd.next());
```

Notice that the global UVM root static class is used here for the context. This mimics the operation by `uvm_root` at simulation start: all `+uvm_set_config_string` plus-args are parsed and set in the `uvm_root` configuration database.

Each UVM component maintains its own resource and configuration database. These values are available via the UVM component functions:

- `set_config_int()` / `get_config_int()`;
- `set_config_string()` / `get_config_string()`; and
- `set_config_object()` / `get_config_object()`.

For each `get()` function call, the local component class is searched for the configuration parameter name, then its parent, etc., up to the root. However, from the static configuration database convenience layer interface, `uvm_config_db`, the global `uvm_root` static instance is searched first. Thus, setting the value within the `uvm_root` ensures it has the highest precedence when the subsequence `uvm_config_db#(T)::get()` function is called.

Returning to the example `bus_env` from section II, we can now set `num_actors` back to a simple integer, remove the control flag, and use the random convenience layer to randomize, as necessary.

```
class bus_env extends uvm_env;
    bus_actor_agent actors[];    // Number determined at sim-time
    int num_actors;  // Valid range is 1-10

    function void build_phase(uvm_phase phase);
        if(!config_rand_db#(int)::get(this, "", "num_actors",
                                          num_actors)) // Store locally
        begin
            `uvm_fatal("CFG_ERR", "Number of actors unknown")
        end
    endfunction
endclass
```

The default number of actors should be a set in a component external to the `bus_env`, and allow the user to override, and optionally randomize, the value on the command line.[2]

```
> simcmd +uvm_set_config_string='*.bus_env,num_actors,inside [1:2]'
```

No class extensions are required to change the random constraint, just a different command line.

[2] Note that some constraint strings require the single-quote, `'`, to avoid unwanted shell interpretation.

```
> simcmd +uvm_set_config_string=\
    '*.bus_env,num_actors,dist{1:=15,2:=50,3:=35}'
```

The constraint may also be specified using the alternative plus-arg approach on the command-line.

```
> simcmd +num_actors='dist{1:=15,2:=50,3:=35}'
```

H. Variation

While the use model for configuration database randomized parameters dictates a single randomization, this need not be so inflexible. Referring to Figure 6 and Figure 7, the `visited` hash may be removed from the flow. In this case, string-typed get from the configuration database occurs each time `config_rand_db#(T)::get()` is called. However, in Figure 7, the command-line search for string constraints still occurs only once. This is appropriate as the command-line constraints cannot change once the simulation begins, whereas string constraints in the configuration database may change. Also, each time `config_rand_db#(T)::get()` is called, in this technique variation, a new instance of `loc_rand#(T)` is instantiated, randomized, and value returned.

V. CONCLUSIONS

Our configuration database random-layer opened the door to command-line random configuration of the simulation. No SystemVerilog random constraints were required within the environment itself. All configuration parameters were made available to randomize from the command-line. From the example in section III:

```
> simcmd +uvm_set_config_string='*.bus_env,num_actors,inside[1:3]'
```

Or, from section IV-F:

```
> simcmd +bus_env.num_actors='dist{1:=15,2:=50,3:=35}'
```

The only source code change required to utilize this technique was to change `uvm_config_db` access to our own convenience-layer class interface, `config_rand_db`. The API and use model within the environment did not change.

We have employed both interchangeable random constraints and the random convenience layer to UVM configuration database in multiple verification projects. These projects support, at the present, both Synopsys VCS [7] and Cadence Incisive© [8] simulators. We have not worked with Mentor Graphics Questa© simulator, but find no reason why it should not work, as well.

REFERENCES

[1] M. Glasser, "Configuration in UVM: The Missing Manual," in *Design & Verification Conference*, Bangalore, India, 2014.

[2] V. R. Cooper and P. Marriott, "Demistifying the UVM Configuration Database," in *Design & Verification Conference*, San Jose, USA, 2014.

[3] Accellera, "Universal Verification Methodology (UVM) 1.1d Class Reference," 2013.

[4] J. Ridgeway, "Interchangeable SystemVerilog Random Constraints," in *Synopsys Users Group*, San Jose, 2014.

[5] P. Goel, A. Sharma and R. Hasija, "Configuring your resources the UVM way!," in *Design & Verification Conference*, San Jose, USA, 2012.

[6] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," New York, 2013.

[7] Synopsys, Inc., VCS Mx / VCS MXi User Guide, H-2013.06-1 ed., 2013.

[8] Cadence Design Systems, Inc., Incisive© Enterprise Simulator, 2014.

APPENDIX

BNF Grammar

```
start: config_variable start
     | config_variable ;

config_variable: config_var ;

config_var: scopename '=' constraint repeat_indicator ;

scopename: stringtoken ':' ':' scopename
         | stringtoken '.' scopename
         | stringtoken ;

repeat_indicator: '*' stringtoken
                | '@' stringtoken
                | ;

constraint: "const" const_value_constraint
          | const_value_constraint
          | "dist" dist_constraint
          | "uniform" uniform_constraint
          | "range" range_constraint
          | "inside" range_constraint
          | "inside" inside_list_constraint ;


// Constraints
const_value_constraint: stringtoken ;

dist_constraint: '{' distList '}' ;

distList: distElem ',' distList
        | distElem ;

distElem: stringtoken ':' '=' stringtoken ;

uniform_constraint: '(' stringtoken ',' stringtoken ')' ;

range_constraint: '[' stringtoken ':' stringtoken ']' ;

inside_list_constraint: '{' insideList '}' ;

insideList: stringtoken ',' insideList
          | stringtoken ;
```