Interchangeable SystemVerilog Random Constraints

Jeremy Ridgeway

LSI Corporation, Inc.
San Jose, CA, USA

www.lsi.com

ABSTRACT

SystemVerilog constraints are declarative in nature. To change the profile on a randomized value for a test, the class containing the constraint is extended and new class instantiated in its place (e.g., via UVM factory), and simulation (re-)compiled. This approach is time- and knowledge-expensive.

We present a suite of SystemVerilog constraint building blocks that are instantiated during simulation. A front-end Flex parser dissects a constraint string, then interacts with VCS through the direct programming interface (DPI) to instantiate a SystemVerilog constraint at random-time on some value (in a type-parameterized container class). A test bench-wide resource manager (e.g., UVM) maintains constraint strings composed before and during simulation.

With the advanced simulation profiler in VCS we show that our constraints are competitive in solving time and, with some optimizations, in simulation time and memory. A significant overhead pervades but flexibility to fully interchange constraints without recompilation is valuable.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

An essential component of constrained random verification (CRV) of hardware is the incorporation of constraints directly in the SystemVerilog language. As noted in [1], "dynamic constraints and randomization" is a requirement for successful CRV. They note that dynamism in SystemVerilog constraints come either through declarative constructs, if-then-else guards in the constraint or in-line constraints (randomize with), or at run-time only through enable/disable. Dynamic randomization is achieved essentially by modifying details of an existing constraint, variables within the constraint, and enable/disable, rather than changing the constraint outright. As a verification engineer we prefer a dynamic method to fully interchange the constraint on-the-fly or even on the command-line.

The Universal Verification Methodology (UVM) library has provided enhanced ability to modify constraints at simulation time via factory overrides. From [2], they note that the UVM factory facilitates "reuse and adjustment of predefined" verification components. This is applicable to any UVM-registered class and therefore applies to constraints. However, to enable factory override the verification engineer must (1) extend the class containing the constraint in question, (2) know the test bench hierarchy to provide the appropriate parameters, and (3) create a new test and recompile. While on-the-fly constraint modifications (i.e., through a function call) would require recompilation in any method, we prefer an easier path.

In section 2, current techniques regarding overriding constraints with SystemVerilog and UVM are discussed. We present our approach to interchangeable constraints in section 3. Then, in section 4 we present the test bench used for results and discussion in section 5.

2. SystemVerilog Constraints

Random constraints may be manipulated either through inheritance (declarative) or enabled or disabled at run-time (dynamic) [3]. Good coding practice defines a default constraint block alongside a declared random value to ensure valid stimulus generation [4]. For example, the constraint in Listing 1 defines the valid length of a packet to be between one and 1K bytes.

```
class packet;
    rand int unsigned length;
    constraint valid {
        length inside {[1:1024]};
    }
    ...
endclass: packet
```

Listing 1: Default constraint is defined alongside its variable declaration.

Test bench components may further constrain the length by extending from the packet class and defining a new constraint. The two constraints are treated as a logical AND and subjected to simultaneous solving (i.e., they must not block all possible values). Also, a class extension may disable the default constraint to allow for invalid stimulus.

Constraint blocks are similar to class member functions in that they are named and, as a language construct, may be overridden through inheritance [3]. Thus, in the event of name-collision, the new constraint completely overrides the inherited constraint.

```
class err_packet extends packet;
    constraint valid {
        length == 0 || length > 1024;
    }
endclass: err_packet

class any_packet extends packet;
    constraint valid { }
endclass: any_packet
```

Listing 2: Override or disable constraints.

In the `err_packet` in Listing 2, the constraint block name *must* be known and overridden to avoid a solving inconsistency, and randomization failure. This becomes a problem when overriding class constraints in encrypted code, such as from a third party vendor, since the constraint block may not be accessible. To declaratively disable a constraint, as in `any_packet`, the name-collided block must be empty.

During simulation, it is possible to enable and disable whole constraint blocks directly via a task [3]. For example, `pkt.valid.constraint_mode(0)` disables the default valid constraint block for the specific packet instantiation, `pkt`. The method is defined by SystemVerilog as both a task and a function. Only the task flavor modifies the current mode on the constraint. Therefore, some care must be taken to modify the mode within the context of a task.

Example: UVM Constraint Override

Both the declarative and dynamic overriding/disabling of SystemVerilog constraints requires knowledge of the test bench hierarchy and the methodology employed. For example, consider the stimulus generation path of the test bench in Figure 1. This test bench complies with UVM [5]. From the figure, the sequence contains a static or dynamic queue of items. Each sequence item is instantiated and populated with constrained random contextual data, such as the packet from Listing 1 or its derivatives.

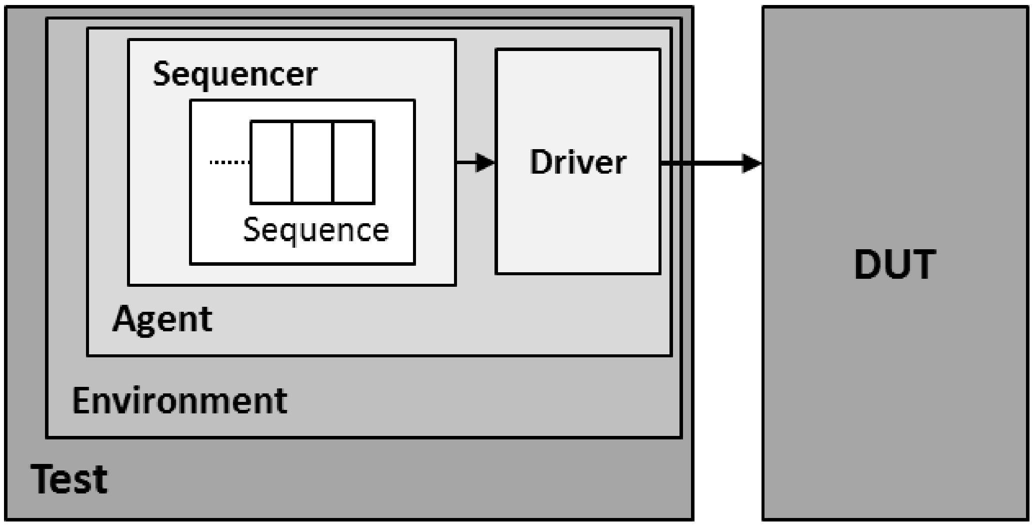

Figure 1: Stimulus generation in UVM.

Assume that some scenario requires transmitting a packet that is exactly 47 bytes in length.

```
class packet47 extends packet;
    `uvm_object_utils(packet47);
    constraint forced_length {
        length == 47;
    }
endclass: packet47

class test_len47 extends uvm_test;
    virtual function build_phase(uvm_phase phase);
        super.build_phase(phase);
        packet::type_id::set_inst_override(
                                      packet47::get_type(),
                                      "env.agent.seqr.seq.*");
    endfunction
    ...
endclass: test_len47
```

Listing 3: Special scenario testing.

In Listing 3, a new packet class is defined along with a new test to register the packet with the UVM library. If not using UVM, then a new sequencer should be defined such that the derived class is instantiated in place of the base class. UVM streamlines this process through its factory [2].

Each UVM object or component class is registered with the UVM factory at compile-time via `` `uvm_object_utils `` or `` `uvm_component_utils `` macros, respectively. Instead of using the `new` operator, the user essentially requests the factory to instantiate the class. If an override has been registered then the factory returns a reference to an instance of the derived class. Overridden classes must be derived because the returned reference must be polymorphic.

UVM provides for class instance and type overriding, each by class type or type name [2]. Listing 3 provides an example of instance by type override. The new class type, `packet47`, is registered with the UVM factory to override instances of `packet` at all test bench paths whose ancestry is denoted in the string.

Finally, once the code modification has been implemented and new class registered with the UVM factory, the test bench must be re-compiled and the new test simulated.

This approach requires a fair amount of detailed test bench knowledge and advanced SystemVerilog and verification methodology understanding. This may be less efficient for quick “what-if” scenarios or sanity checks (e.g., for bug fixes).

3. Interchangeable Constraints

We have defined a set of general classes in a local verification methodology (LVM) library to implement a subset of the SystemVerilog constraint language. When coupled with a front-end parser and test bench-wide resource manager, constraints may be fully interchanged on-the-fly. Furthermore, neither deep test bench architecture nor verification methodology need be known to change the constraints.

Consider the block diagram in Figure 2. Instead of specifying “rand” on a SystemVerilog integral type declaration, we instantiate a new parameterized class, `lvm_rand`. In step 1 of the figure, a call to function `lvm_rand.push(“inside[100:1000]”)` sets in motion a sequence of events that leads to the construction of the `lvm_range` class. This class implements the inside range constraint.

Continuing through the figure, a static and parameterized constraint factory is accessed for actual constraint construction. In step 2, the constraint factory instantiates, through the direct programming interface (DPI) layer, a parser that then generates an abstract syntax tree (AST). The parser implements a subset of the SystemVerilog constraint language and utilizes a flex generated lexer to identify keywords in the string and a bison generated parser to build the AST [6, 7]. [1, 2, 3] Once parsing is complete, the AST reflects the structure of the constraint such that the constraint factory may, in step 3, instantiate it in SystemVerilog. Finally, a well-formed constraint is returned to the original `lvm_rand` instance in step 4.

[1] For fast lexical analyzer details, see: http://en.wikipedia.org/wiki/Flex_lexical_analyser.
[2] For GNU bison details, see: http://en.wikipedia.org/wiki/GNU_bison.
[3] Refer to the appendix for the interchangeable constraints Backus-Naur Form (BNF) grammar.

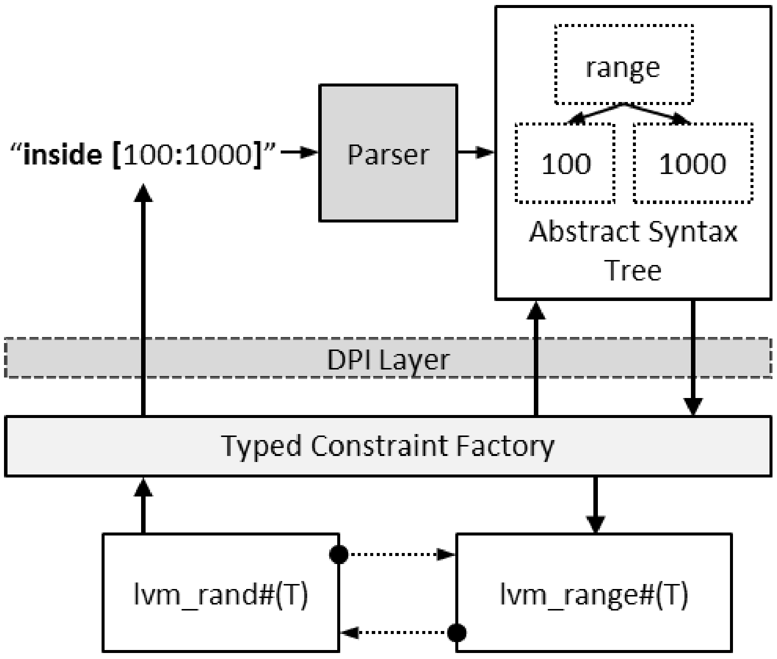

Figure 2: A new interchangeable constraint.

SystemVerilog class hierarchies are shown in Figure 3 for LVM random variables and constraints. The vertical arrows in the figure indicate class inheritance. A typed base class, `lvm_param_rand`, is extended from an untyped base class for flexibility in reference comparison. Also, references to a factory and its current constraint (constructed by the factory) exist in the typed base class. Because the constraint factory is re-entrant and memory-less, its reference is attributed "static" to limit the number of instances (one per type).

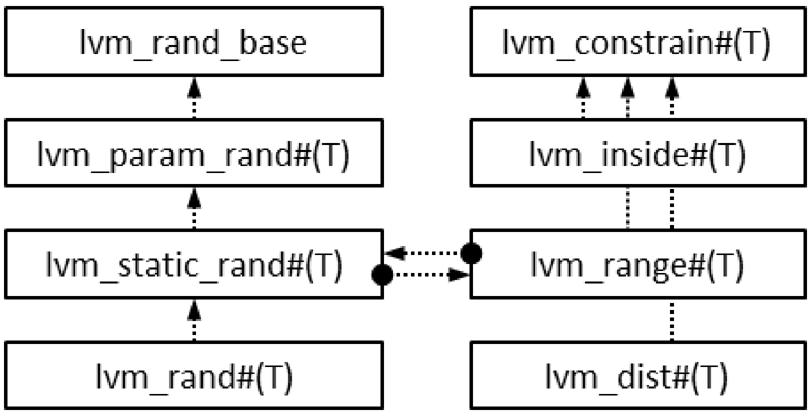

Figure 3: Random and constraint hierarchies.

The typed SystemVerilog random variable is declared within `lvm_static_rand`. This class is "static" in that its constraint may only be manipulated via the public push/pop API. The next function returns the randomized value. However, as the value is not protected, user code may access it directly.

```
class lvm_param_rand#(T) extends lvm_base_rand;
    protected lvm_constrain#(T) m_curr_constraint;
    static protected
        lvm_constraint_factory#(T) m_factory;
endclass

class lvm_static_rand#(T) extends lvm_param_rand;
    rand T value;
    virtual function void push(string n);
    virtual function void pop();
    virtual function T next();
endclass

class lvm_rand#(T) extends lvm_static_rand;
    function void m_update_constraints();
    virtual function T next();
endclass
```

Listing 4: Random variable use-classes.

Finally, the `lvm_rand` class accesses the test bench-wide resource manager. While the manager may be any table-style lookup, we employ the resource database in the UVM library. The interchangeable constraints language considers test bench scoping to pinpoint or globalize the application of new constraints.

The randomization function, next, is detailed in Algorithm 1. This function is used as a differentiating device between interchangeable and SystemVerilog random variables. In the function, first the current constraint is updated if a new string exists in the UVM resource database. Because that string is cleared immediately after use, if a string exists then that indicates the constraint must be interchanged. However, in line 5, when the current constraint is null, the value is randomized unconstrained. The current constraint is null only at construction and as the result of the pop function. Thus, constraints may be disabled on-the-fly by the user by simply popping them. Lines 1-4 do not exist for `lvm_static_rand`, so explicit interaction is required to interchange constraints.

Algorithm 1: Next random value generation.

```
1: if new constraint exists in uvm_resource_db#(string) then
2:     m_curr_constraint ← push(constraint_string)
3:     clear string in uvm_resource_db
4: end if
5: if m_curr_constraint ≠ null then
6:     value ← m_curr_constraint.next()
7: else // value is unconstrained
8:     value ← randomize()
9:     if m_curr_constraint is done then
```

```
10:         m_curr_constraint ← null
11:     end if
12: end if
```

Our interchangeable constraints implement constants, inside ranges, inside lists, and user defined distribution lists. Constraints require a reference to an `lvm_static_rand` instance or derivative, as indicated by the left directional arrow in Figure 3.

```
class lvm_constrain #(type T=int);
    protected lvm_rand_static#(T) loc_rand;
    pure virtual function T next();
endclass

class lvm_const #(type T=int) extends lvm_constrain#(T);
    local const T const_value;
    function T next();
        loc_rand.randomize with { value == local::const_value};
    endfunction
endclass
```

Listing 5: The constraint base class references the random variable and defines the randomization interface. The extension class defines and use members necessary to randomize.

Each constraint type is a separate SystemVerilog class that is extended from the `lvm_constrain` base and maintain necessary members. Then, an in-line randomize call is employed using those members and targeting the local `lvm_static_rand` reference. Constant constraints, `lvm_const`, have a single typed value to use in randomization, as in Listing 5. Inside ranges, `lvm_range`, have a minimum and maximum typed value; inside lists, `lvm_inside`, have a typed queue of values; distributions, `lvm_dist`, have a typed queue of values and an integer queue of weights.

Randomization follows similar to `lvm_const` for all constraints except for the distribution. We employ a common linear two-step strategy for user-defined distributions. First, the chosen weight is uniformly randomized in the range [1:*total_weight*]. Then, the chosen weight corresponds to the chosen value by iterating through the weight queue as in Algorithm 2. The time required to determine the chosen value queue slot grows linearly with the number of elements ($O(n)$). The grammar in the appendix defines range as an alias to inside range and uniform as a Boolean variant (i.e. $min \le value \le max$ rather than inside [*min*:*max*]).

Algorithm 2: User-specified distribution constraint.

```
1: curr_wt ← randomize { inside [1:total_weight] }
2: for all wt_queue[i] do
3:     nxt_wt_sum ← nxt_wt_sum + wt_queue[i]
4:     if wt_sum < curr_wt ≤ nxt_wt_sum then
5:         loc ← i
6:         break
```

```
 7:     end if
 8:     wt_sum ← nxt_wt_sum
 9: end for
10: loc_rand.value ← randomize { value == val_queue[loc] }
```

The constraint factory, from Figure 2, links a flex generated C++ lexer and bison generated C++ parser to SystemVerilog via the DPI. Using C++ allows for multiple instances of the lexer and parser to coexist [6, 7]. At step 2 in the figure, a single structure encapsulating lexer, parser, and all necessary data to build an AST from the input string is allocated in C++ and stored on the SystemVerilog runtime stack as an automatic `chandle` type. The lexer and parser build the AST in C++ while instantiation of the type-specific constraint occur in SystemVerilog. Algorithm 3 presents the general algorithm for stepping through an AST and constructing a new SystemVerilog constraint container class.

Algorithm 3: Constraint factory interchangeable constraint construction.

```
 1: lvm_constrain#(T) new_c ← null
 2: chandle AST ← parse_begin(constraint_string) // DPI function call
 3: switch get_constraint_type(AST) do // DPI function call
 4:     case const_constraint:
 5:         lvm_const#(T) const_c;
 6:         uvm_bitstream_t val;
 7:         T lval;
 8:         val ← get_next_value(AST); // DPI function call
 9:         cast { lval ← val  }
10:         const_c ← new(lva );
11:         cast { new_c ← const_c };
12:     end
13:     … // follows similarly for remaining constraint types
14: endsw
15: parse_end(AST) // DPI function call, release memory
16: return new_c
```

Note that both the interchangeable constraint and the constraint factory are type-parameterized. While the constraint is dynamic and can change throughout simulation, the type is static. It is not legal to change the random variable from one type to another. Furthermore, SystemVerilog supports direct randomization of integral types only [3]. Interchangeable constraints also support SystemVerilog types only. This implies that interchangeable constraints may not be nested. For example, the following code is illegal.

```
lvm_rand#(lvm_rand#(int)) my_complicated_rand; // Illegal !!
```

Also note, the translation from C++ data structure to `uvm_bitstream_t` is realized in SystemVerilog. We ran into issue with UVM regular expressions leaking memory during simulation and therefore implemented a two step approach. First, the type and arguments to the

SystemVerilog value is classified with a front-end lexer returning a format string, `fmt`. Then, conversion actually occurs on the entire value in SystemVerilog:

```
uvm_bitstream_t v = $sscanf(fmt, value);
```

Employing SystemVerilog system functions reduce memory consumption and avoided the UVM library.

The entire process from lexical analysis to constraint construction is handled solely within non-blocking SystemVerilog and C++ functions. At completion, the `chandle` is freed, leaking no memory (although refer to the analysis in section 6). These measures ensure interchangeable constraint factories are re-entrant and, therefore, may be statically instantiated.

The interchangeable constraints are lazy. Nothing occurs until and only if it is required. For example, if a constraint string is listed on the command-line, it will only be parsed when and if the associated `lvm_rand` calls its next function. Thus, constraints do not exist unless they are explicitly necessary. Further, each class or function in the constraint interchange process only handles what is required. For example, the AST generated from parsing is not an exact model of the constraint because number conversion is not handled in the parser, this task occurs later in SystemVerilog.

4. Test Case

As a test case of interchangeable random constraints, we implemented two packet generation only test benches conforming to UVM, similar to Figure 1 (i.e. UVM sequence, sequencer, driver, and agent). The two test benches differed in the base packet definition used in the sequence:

- Test bench #1 used SystemVerilog constraints, as in Listing 6,
- Test bench #2 employed interchangeable constraints, as in Listing 7.

Notice that in Listing 6 the default constraint defined the entire valid range, as expected.

```
class packet extends uvm_sequence_item;
  rand int unsigned length;
  `uvm_object_utils(packet);
  constraint valid { length inside {[0:4096]}; }
endclass: packet

class scenario2 extends packet;
  `uvm_object_utils(scenario2);
  constraint scen { length inside {[1:1023]}; }
endclass: scenario2

class err_toobig extends packet;
  `uvm_object_utils(err_toobig);
  constraint scen { length > 4096; }
```

```
endclass: err_toobig
```

Listing 6: Packet types with SystemVerilog constraints.

The constraint block in `scenario2` further constrained the packet length within the valid space while `err_toobig` forced the packet length beyond the valid space. A common error is to inadvertently define an inconsistent constraint, especially through inheritance. Without disabling or overriding the default constraint, `err_toobig` will fail.

In contrast, the interchangeable versions of all packets are defined in just the base packet.

```
class packet extends uvm_sequence_item;
  `uvm_object_utils(packet);
  lvm_rand#(int unsigned) length;
  function new(string name = "packet");
    super.new(name);
    length = new("PKT_LENGTH", this);
    length.push("inside[0:4096]");
  endfunction
endclass: packet
```

Listing 7: Packet with interchangeable constraints.

In Listing 7, length is declared as an `lvm_rand` and instantiated at packet construction with the name `PKT_LENGTH`. The default valid range is pushed onto the variable. At randomization time, the default constraint may be overridden by a new shape existing in the UVM resource database or from the command-line. Since the new constraint will fully replace the existing one, solving consistency is based only on the new constraint string.

Following UVM guidelines, we generated packets in the sequence in a loop. The number of packets to be generated was controlled in the test via an `uvm_config_db` parameter. In the packet sequence, we generated packets in a loop using the standard UVM macro:

```
virtual task body();
  if(starting_phase != null)
    starting_phase.raise_objection(this);

  for(int i=0; i<num_pkts; ++i)
    `uvm_do(req)

  if(starting_phase != null)
    starting_phase.raise_objection(this);
endtask
```

Listing 8: Sequence body task using standard `` `uvm_do `` approach.

For test bench #2, the interchangeable constraint was set in the resource database for regression testing. We also ran the same test using the uvm_config_string command-line plusarg. Each test case in the results section shows both methods.

5. Results

We simulated with VCS version 2013.06-1 full 64-bit mode on shared Linux machines with 32 Intel (R) Xenon(R) E5-2690 processors at 2.90GHz and 378 GB of memory [8]. For the performance results section, we enabled Synopsys VCS's Unified Simulation Profiler, a limited customer agreement option [9].

For both test benches, we simulated a single constraint in each of three tests:

- Test 1: Distribution,
- Test 2: Inside set,
- Test 3: Inside range.

Each test was executed five times, randomizing the variable according to the constraint a fixed number of times: 500; 5,000; 50,000; 500,000; and 5,000,000 randomizations. The random distributions observed are presented first; while the resultant performance metrics are presented second.

Distribution Results

In test 1 for test bench #2, the inside set constraint was simulated. The interchangeable constraint was set in a test through the UVM resource database:

```
uvm_resource_db#(string)::
    set_override("*", "PKT_LENGTH",
           "dist{1:=1, 256:=2, 512:=2, 1024:=3,
            1280:=3, 1536:=3, 1792:=3, 2048:=1, 2304:=1,
            2560:=1, 2816:=1, 3072:=2, 3328:=2, 3584:=2,
            3840:=2, 4096:=1 }", this);
```

The following command-line argument is equivalent (spacing is for readability, they were not used in code):

```
+uvm_config_string="PKT_LENGTH","dist{1:=1, 256:=2, 512:=2,
            1024:=3, 1280:=3, 1536:=3, 1792:=3, 2048:=1,
            2304:=1, 2560:=1, 2816:=1, 3072:=2, 3328:=2,
            3584:=2, 3840:=2, 4096:=1 }".
```

Test bench #1 implemented the corresponding SystemVerilog constraint block. Figure 4 shows the observed behavior for both test bench #1, SystemVerilog constraints, and test bench #2, interchangeable constraints. As expected, as the number of randomizations increase, the two distributions converge to an identical line.

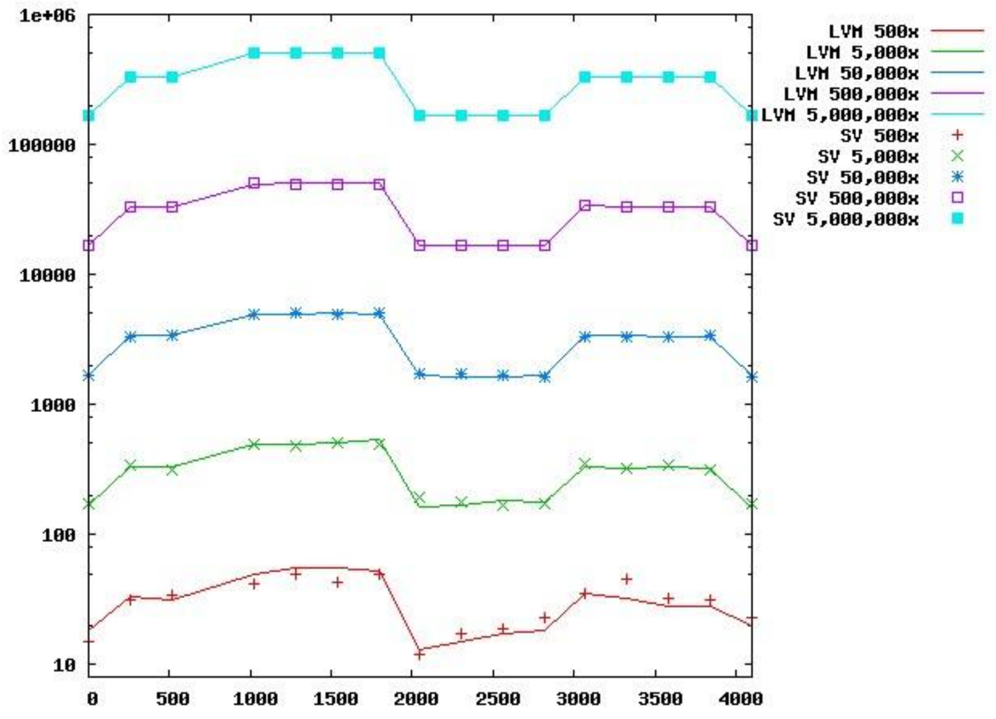

Figure 4: User defined distribution: "dist".

In test 2 for test bench #2, the inside set constraint was simulated. The interchangeable constraint was set in a test through the UVM resource database:

```
 uvm_resource_db#(string)::
     set_override("*", "PKT_LENGTH", "inside{1, 256, 512, 1024,
                  1280, 1536, 1792, 2048, 2304, 2560,
                  2816, 3072, 3328, 3584, 3840, 4096 }", this);
```

The following command-line argument is equivalent:

```
 +uvm_config_string="PKT_LENGTH","inside{1, 256, 512, 1024,
                     1280, 1536, 1792, 2048, 2304, 2560,
                     2816, 3072, 3328, 3584, 3840, 4096 }".
```

Test bench #1 implemented the corresponding SystemVerilog constraint block. Figure 5 shows the observed behavior for both test benches. As expected, as the number of randomizations increase, the two distributions converge to an identical line.

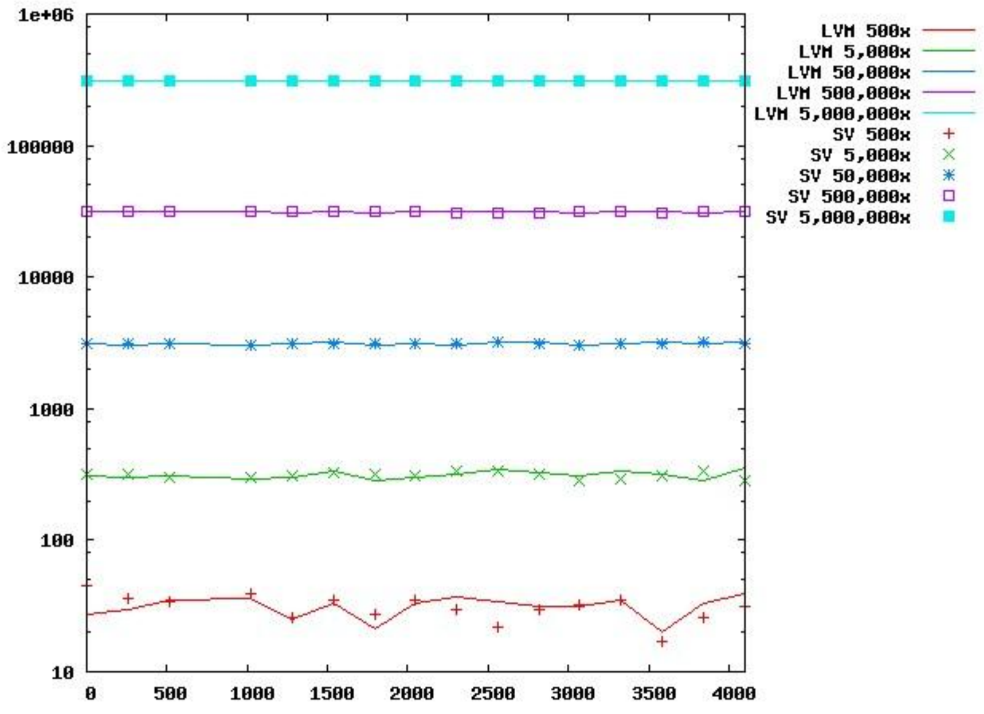

Figure 5: Uniform distribution on a set of distinct values: "inside".

In test 3 for test bench #2, the inside range constraint was simulated. The interchangeable constraint was set in a test through the UVM resource database:

```
uvm_resource_db#(string)::
    set_override("*", "PKT_LENGTH", "inside [30:50]", this);
```

The following command-line argument is equivalent:

```
+uvm_config_string="PKT_LENGTH","inside [30:50]"
```

Test bench #1 implemented the corresponding SystemVerilog constraint block. Figure 6 shows the observed behavior for both test benches. As expected, as the number of randomizations increase, the two distributions converge to an identical line.

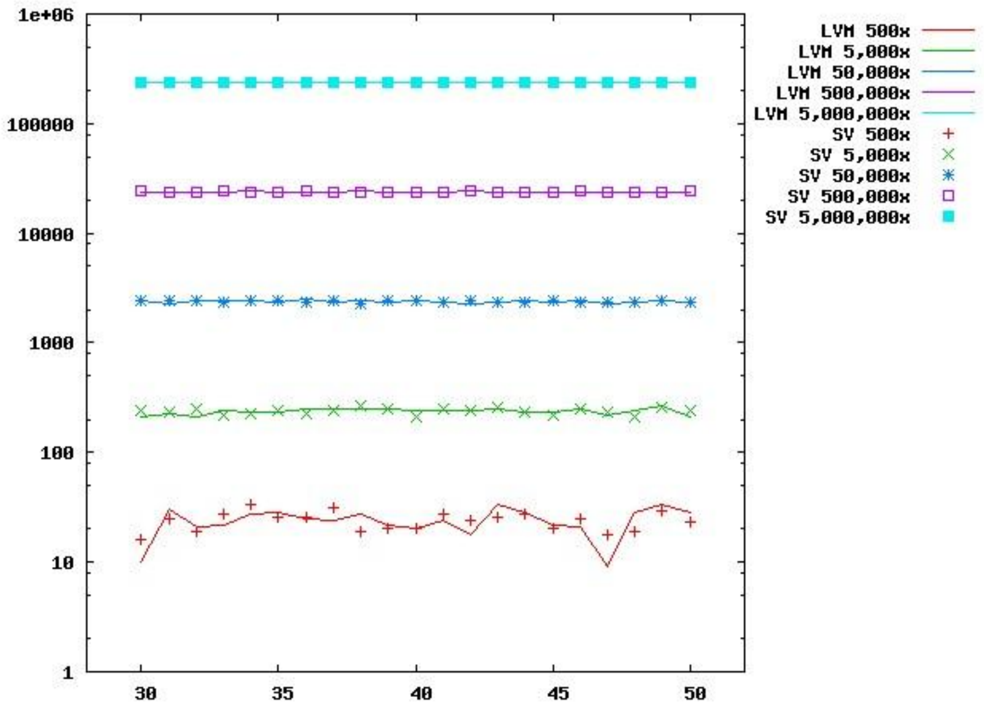

Figure 6: Uniform distribution on a range of values

Performance Results

For performance comparison, we considered the following metrics:

- Cumulative simulation time,
- Cumulative simulation memory,
- Constraint solving time,
- Constraint solving memory.

Advanced simulation profiling in VCS was enabled for constraint metrics. For the cumulative simulation metrics, only a summary report was generated. On the command line, we used the VCS runtime option `-reportstats`, for CPU time and memory usage (virtual memory size) at simulation completion.

```
Simulation Performance Summary
==============================
Simulation started at :  Fri Dec  6 07:23:19 2013
Elapsed Time           :  8 sec
CPU Time             :  4.2 sec
Virtual memory size  :  174.7 MB
Resident set size    :  93.8 MB
Shared memory size     :  22.9 MB
Private memory size    :  70.9 MB
Major page faults      :  70
```

```
==============================
```

As mentioned in section 1, interchangeable constraints were implemented as part of a local verification methodology (LVM) library built atop UVM. Care was taken in test bench #2 to avoid any additional calls to that library that didn't already exist in test bench #1.

Regression with uvm_do

The first round of regression cumulative simulation metrics are reported in Table 1. The large discrepancy between the two techniques is obvious in this data set; SystemVerilog constraints are the clear winner. However, in Table 2, the results are not so bleak for interchangeable constraints. While SystemVerilog is still the winner in time and memory use, the gap is much narrower. Investigations into why the discrepancy is so large between the two techniques lead us to some optimizations; see the next section for details and results.

Table 1: Cumulative simulation performance using `` `uvm_do ``.

Test	TB #1 SysV Time (s)	TB #1 SysV Mem (MB)	TB #2 lvm Time (s)	TB #2 lvm Mem (MB)
t1_5000000	899.1	589.3	11186.9	249682.4
t1_500000	77.4	174.3	483.8	5917.6
t1_50000	7.9	155.0	42.7	400.0
t1_5000	1.1	151.3	4.2	174.7
t1_500	0.5	148.3	0.8	156.2
t2_5000000	832.0	510.8	6661.1	218433.2
t2_500000	68.4	177.7	351.2	4959.3
t2_50000	7.1	154.6	30.4	313.2
t2_5000	1.1	151.3	3.8	166.2
t2_500	0.4	148.3	0.6	155.3
t3_5000000	814.0	441.1	3183.8	247283.6
t3_500000	69.7	176.3	195.0	4354.7
t3_50000	7.2	154.1	17.8	258.6
t3_5000	1.1	151.3	2.0	160.0
t3_500	0.4	148.3	0.5	153.7

Table 2: Constraint solver performance using `` `uvm_do ``.

Test	TB #1 SysV Time (s)	TB #1 SysV Mem (KB)	TB #2 lvm Time (s)	TB #2 lvm Mem (KB)
t1_5000000	214.430	176	364.030	40
t1_500000	21.140	176	32.090	40
t1_50000	2.460	176	2.680	40
t1_5000	0.230	176	0.310	40
t1_500	0.010	176	0.020	40
t2_5000000	152.280	40	265.650	72
t2_500000	15.830	40	26.360	72

t2_50000	1.470	40	2.780	72
t2_5000	0.230	40	0.310	72
t2_500	0.040	40	0.040	72
t3_5000000	148.800	24	162.010	16
t3_500000	14.930	24	14.750	16
t3_50000	1.340	24	1.750	16
t3_5000	0.160	24	0.060	16
t3_500	0.020	24	0.030	16

Regression with optimizations

We identified two bottlenecks in the interchangeable constraint technique, neither related to solving:

1. Packet instantiation in `` `uvm_do `` macro, and
2. Resource database pattern matching.

First, the standard `` `uvm_do `` macro approach, as shown in Listing 8, instantiates a new sequence item for each loop [5]. During large simulations (e.g. 500,000 packets or more), memory consumption seems to jump considerably and inversely to performance. Therefore, we modified the sequence to instantiate the packet once, a *persistent instance*, and randomize only each loop.

Second, the UVM resource database performs glob or regular expression pattern matching during lookup. This enables wildcard use in test bench path arguments [2]. We took two corrective actions to optimize the regression. We internally replaced the UVM resource database with a very simple associative array, a *local constraint database*, keyed only on the random variable name (i.e. `lvm_rand.get_name()`).

```
constraint_db["name"] = "interchangeable constraint"
```

Then, we automatically disabled interchangeable dynamism following first randomization, so-called *single dynamism*. The `lvm_rand` class provided this API already:

```
lvm_rand#(int) my_rand;
my_rand.clear_dynamic();
```

These changes effectively disabled uvm_resource_db access for setting constraints (replaced by another global API) and allowed only a single dynamic update on the constraint, for the purpose of testing.

In additional regressions, we incrementally optimized bottlenecks, as described above. Three sets of regressions were run:

Optimization 1 – uvm_resource_db + persistent instance,

Optimization 2 – uvm_resource_db + persistent instance & single dynamism, and

Optimization 3 – local constraint database + persistent instance & single dynamism.

The optimization regressions cumulative simulation metrics are reported in Table 3 and Table 4. To be fair, optimization #1 was implemented in the SystemVerilog constraints test bench. The remaining optimizations have no effect on SystemVerilog constraints.

The first set of regressions using `` `uvm_do ``, in Table 1, did not seem to follow a logarithmic path even though the number of randomizations scaled accordingly. However, the time signatures in Table 3 are obviously logarithmic. Furthermore, once optimization #2 is employed, there is very little overall difference between the SystemVerilog constraints and interchangeable constraints.

Table 3: Cumulative simulation performance over successive optimizations.

Test	**TB #1 SysV** *Optimization 1* **Time (**s**)**	**TB #2 lvm** *Optimization 1* **Time (**s**)**	**TB #2 lvm** *Optimization 2* **Time (**s**)**	**TB #2 lvm** *Optimization 3* **Time (**s**)**
t1_5000000	613.9	2949.9	766.6	762.7
t1_500000	60.4	270.3	80.6	76.3
t1_50000	6.2	28.3	7.9	7.9
t1_5000	1.0	3.1	1.1	1.1
t1_500	0.4	0.6	0.4	0.5
t2_5000000	527.3	2809.9	659.8	681.6
t2_500000	54.2	258.1	69.0	65.9
t2_50000	5.7	26.3	6.9	7.5
t2_5000	0.9	2.9	1.1	1.0
t2_500	0.4	0.6	0.4	0.4
t3_5000000	537.8	2634.8	548.6	550.1
t3_500000	52.2	244.7	56.0	54.6
t3_50000	5.7	25.2	6.0	5.7
t3_5000	0.9	2.8	0.9	1.0
t3_500	0.4	0.6	0.5	0.4

Table 4: Cumulative simulation memory consumption over successive optimizations.

Test	**TB #1 SysV** *Optimization 1* **Mem (**MB**)**	**TB #2 lvm** *Optimization 1* **Mem (**MB**)**	**TB #2 lvm** *Optimization 2* **Mem (**MB**)**	**TB #2 lvm** *Optimization 3* **Mem (**MB**)**
t1_5000000	151.3	3664.2	153.3	153.3
t1_500000	151.3	547.7	153.3	153.3
t1_50000	151.3	198.2	153.3	153.3
t1_5000	148.3	154.4	150.3	150.3
t1_500	148.3	151.4	150.3	150.3
t2_5000000	151.3	3681.5	153.3	153.3
t2_500000	151.3	549.9	153.3	153.3
t2_50000	151.3	198.2	153.3	153.3
t2_5000	148.3	154.4	150.3	150.3
t2_500	148.3	151.4	150.3	150.3

t3_5000000	151.3	3654.6	154.3	174.6
t3_500000	151.3	549.0	154.3	153.3
t3_50000	151.3	219.5	150.3	153.3
t3_5000	148.3	154.4	150.3	150.3
t3_500	148.3	172.8	150.3	150.3

6. Discussion and Future Work

The allocated memory in C++ is always freed following constraint construction. In Table 4 optimization #1 should have followed optimizations #2 and #3 since just one packet was instantiated in the simulation. This is clearly not the case. Because optimization #2 uses the UVM resource database, the fault does not necessarily lie in UVM. Likely, our parser implementation and/or the DPI linkage is not properly releasing or properly calling to release the memory. The result is an explosion of space. Therefore, if millions of randomizations are to occur, then interchangeable constraints should be limited to simulation configuration time (i.e., UVM pre-run phases). Restricting to single dynamism allows constraint override once through API or command-line prior to randomization. Future work remains to further optimize the DPI linkage.

Some limitations on interchangeable constraints exist in the current implementation. First, as seen in Algorithm 3, each of the string values is cast to a SystemVerilog integral type (and agreeing with the `lvm_rand` type). Referencing a parent class variable is not legal. Furthermore, nesting of constraint strings is not legal as is most mathematical and Boolean operations. The following cases are not currently supported by interchangeable constraints.

```
class a;
  int min = 4;
  lvm_rand#(int) my_val;
  function new();
    my_range = new(“MY_RANGE”);
    my_val.push(“inside [min:10]”); // Illegal!
                               // Parent class variable access.
    my_val.push(“inside [inside [0:10] : inside [100:200]]”);
                               // Illegal! Nested constraints.
    my_val.push(“value > 4”); // Illegal! Math/logical op.
  endfunction
endclass
```

Listing 9: Current interchangeable constraint limitations.

Many times dynamic composition of the constraint string will suffice in overcoming limitations. For example, instead of using parent class variables within the constraint, use them to *compose* a new constraint string.

```
my_val.push($sformatf(“inside [%0d:10]”, min)); // Legal
```

Accessing the current local class variable values from within the constraint itself remains an issue that is not overcome. However, there is no direct SystemVerilog limitation on mathematical and logical operation support. This is a focus of future work.

7. Conclusions

We presented an approach that enables fully interchangeable SystemVerilog constraints by defining a set of general-use classes coupled with a front-end parser and test bench-wide resource manager. Overall, our approach provides a straightforward way to specify constraints as a string either to a function or on the command line. We demonstrated the resultant distributions, in the macro, converge with SystemVerilog native constraints. However, interchangeable constraints may incur a significant overhead depending on the use model. When properly optimized, however, interchangeable constraints are competitive with SystemVerilog constraints with an important advantage: constraints can be entirely swapped during simulation.

8. References

[1] J. Yuan, C. Pixley and A. Aziz, Constraint-Based Verification, New York: Springer Science+Business Media, Inc., 2006.

[2] Accellera, "Universal Verification Methodology (UVM) 1.1 User's Guide," 2011.

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

[4] C. Spear, SystemVerilog for Verification, A Guide to Learning the Testbench Language Features, New York: Springer Science+Business Media, LLC, 2010.

[5] Accellera, "Universal Verification Methodology (UVM) 1.1 Class Reference," 2012.

[6] V. Packson, "Flex, A Fast Scanner Generator," March 1995. [Online]. Available: http://flex.sourceforge.net.

[7] C. Donnelly and R. Stallman, "Bison, The Yacc-compatible Parser Generator," 2 April 2009. [Online]. Available: http://www.gnu.org/software/bison/bison.html.

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

[9] Synopsys, Inc., VCS Mx / VCS MXi LCA Features, H-2013.06-1 ed., 2013.

9. 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 ;
```