Synopsys Users Group

Complex Constraints – Unleashing the Power of the VCS Constraint Solver

John Dickol

Samsung Austin R&D Center

Austin, TX

ABSTRACT

Recent advances in VCS constraint solver performance make it practical to use more complex constraints in verification environments. This paper will review some lesser-known SystemVerilog constraint features (soft constraints, hierarchical constraints, array reduction constraints) and provide working examples of using these and other features to create constrained stimulus for complex real-world CPU/SOC verification problems such as testbench configuration, memory map allocation, transaction address selection, etc. Examples will be given for both UVM and non-UVM environments. The paper will also discuss techniques for debugging constraints (solver failures and failure to generate desired results.)

Table of Contents

Table of Figures

1. Introduction

SystemVerilog [1] constraints provide a powerful and expressive mechanism for controlling random stimulus. As constraint solvers evolve, they are capable of solving more complex constraint problems. The intent of this paper is to show how to harness the power of the constraint solver and apply it to real-world verification problems

This paper starts with a review of some new or lesser-known SystemVerilog constraint techniques. It then provides examples of using these techniques to solve typical complex verification problems. Finally, it provides a brief overview of constraint debug techniques available in Synopsys VCS.

2. Review of SystemVerilog Constraint Techniques

Each revision of the SystemVerilog standard has added new constraint capabilities. This section reviews and gives brief examples of some of the newer or infrequently-used constraint techniques now available in the language.

2.1 Soft Constraints

"Soft" constraints, added to the language in 2012, are constraints which apply only if they are not contradicted by other constraints. They are useful for specifying default values or typical ranges in a class. These defaults may be overridden by hard (non-soft) constraints or by soft constraints with a higher priority. In general, SystemVerilog defines soft constraint priorities such that the last constraint specified has higher priority. For example, inline constraints (i.e. randomize with {…}) have a higher priority than constraints specified in the object being randomized. Refer to [1] for a full list of priority rules for soft constraints.

Figure 1 shows a typical use of soft constraints in a Universal Verification Methodology (UVM) [2] sequence to specify a default range for the number of transactions to be generated. (Note: some UVM boilerplate code has been omitted in this example for simplicity). The sequence (myseq) uses a soft constraint to specify a default range for "count" (the number of transactions generated by the sequence.) A higher-level sequence (topseq) runs myseq and uses an inline constraint to override count. Without soft constraints, this would result in a constraint solver error since count cannot be between 10 and 20 and also equal to 1000. Because the (hard) inline constraint has a higher priority than the soft constraint, the soft constraint is discarded.

```
class myseq extends uvm_sequence#(mytxn);
  rand int count // number of transactions generated by the sequence

  constraint c_count {
    soft count inside {[10:20]};
  }

  task body;
    mytxn  txn;   // transaction generated by this sequence

    repeat(count) begin
      `uvm_do(txn)
    end
  endtask
endclass

class topseq extends uvm_sequence#(mytxn);

  task body;
    myseq  seq;
    // default constraint – generates 10-20 transactions
    `uvm_do(seq)

    // “with” constraint overrides soft constraint in myseq
    `uvm_do_with(seq, {count == 1000;}
  endtask
endclass
```

Figure 1. Soft constraint specifying default range in UVM sequence

2.2 Unique Constraints

The “unique” constraint keyword was added to the language in 2012. It constrains a set of values (either scalar integral variables or unpacked arrays of integral values) to all have unique values. Arrays and scalars can be mixed in the set of variables. Rand and non-rand variables can be used (but not randc);

Figure 2. shows some simple uses of “unique”.

```
rand int x, y, z;
rand int arr[10];
     int q[$];     // Non-rand queue

constraint c_unique {
  unique {x, y, z};   // equivalent to: x!=y; y!=z; z!=x;
  unique {arr};       // Fill arr with 10 unique values
  unique {x, y, arr}  // Can mix scalars and arrays
  unique {x, y, q};   // pick x & y not equal to non-rand queue values
}
```

Figure 2. Example of unique constraints

2.3 Array Constraints

Constraints can be applied to each element of an array (including queues and associative arrays) using a "foreach" iterative constraint. It's also possible to constrain the size of a variable-length array or queue by using the array's size method. Per the SystemVerilog Language Reference Manual (LRM), size constraints are solved before the iterative constraints. Depending on the iterative constraints used, this could cause a constraint solver failure.

Figure 3 shows some simple examples of array constraints. Note the use of the array index (i) in the sorted queue constraint. It is used to prevent the creation of a constraint for the case where i=0. Without this "constraint guard", the foreach loop would create a constraint q[0]<q[0-1] which would produce an index out of bounds error.

```
rand int arr[10];
rand int   q[$];

constraint c_foreach {
  foreach(arr[i])  arr[i] inside {[0:15]};

  q.size() inside {[5:10]};  // constrain queue size
  foreach(q[i]) {
    if(i>0) q[i] > q[i-1];   // create sorted queue: q[0]<q[1]<q[2]...
  }
}
```

Figure 3. Array constraints

2.4 Array Reduction Constraints

In addition to the size method, array variables have several other built-in methods (sum, product, and, or, xor) which may be used in constraints. These methods apply the specified operation between each element of the array:

```
rand int a[10];
a.sum     == a[0] + a[1] + …
a.product == a[0] * a[1] * …
a.and     == a[0] & a[1] & …       // bitwise AND
a.or      == a[0] | a[1] | …       // bitwise OR
a.xor     == a[0] ^ a[1] ^ …       // bitwise Exclusive OR
```

Reduction methods may also be used on multi-dimensional arrays in which case the specified operation applies to all elements over all array dimensions:

```
rand int b[3][4];
b.sum == b[0][0] + b[0][1] + …
         b[1][0] + b[1][1] + …
         …
```

By default, these methods return a single value of the same type as each array element, i.e. the sum of an int array returns a single int, the xor of a bit array returns a single bit, etc. This behavior can be changed by adding a "with" expression which is applied to each array element before applying the reduction operation. If a "with" expression is used, the method return type is the type of the "with" expression. Inside this expression, the special variable "item" can be used to refer to the current array element in the iteration.

A common application of "with" expressions is constraining the number of 1s in an array of bits. Without a "with" expression, the result of the sum method would be a single bit which is not large enough to contain the actual sum. By casting each element to a larger size (e.g. int), the result will be large enough to contain the actual sum.

The array index corresponding to the current "item" is available via the "index" method of the iterator variable (i.e. "item.index"). For multi-dimensional arrays, the optional argument to the index method specifies which array dimension is wanted. An argument of 1 (the default) specifies the first (leftmost) dimension, 2 (i.e. "item.index(2)") specifies the second dimension, etc.

Figure 4 shows some simple examples of array reduction constraints.

```
rand bit bq[$];
rand int a[10];

constraint c_sum {
  bq.size inside {[10:20]};
  bq.sum with (int'(item)) == 7;  // Cast to int before adding

  a.sum with ((item.index < 3) ? item : 0);  // sum 1st 3 elements of a
  // sum with (…) equivalent to:  foreach(a[i]) if(i<3) sum+= a[i];
  // item equivalent to:          a[i]
  // item.index equivalent to:    i
}
```

Figure 4. Array reduction constraints

2.5 Hierarchical (Global) Constraints

A class with rand variables and constraints may be a rand member of another class. When the top-level class is randomized, all rand variables and constraints in the top- and lower-level classes are solved simultaneously. The top-level object may constrain variables in the lower-level objects. The SystemVerilog LRM calls these "global constraints" although a more intuitive term might be "hierarchical constraints."

Figure 5 shows a simple example of this for a simple line-drawing application. A class (XY) is used to represent the coordinates of a single point. It has constraints to limit the X and Y coordinates to legal values. A second class (Line) instantiates two rand instances of the XY class and adds additional constraints to limit the line to being either horizontal or vertical. Randomizing an instance of Line will simultaneously solve the constraints for the two rand XY points and the additional Line direction constraints.

```
class XY;
  rand int x;
  rand int y;

  constraint c_valid {
    x inside {[0:100]};
    y inside {[0:100]};
  }
endclass

class Line;
  rand XY                          pt1;
  rand XY                          pt2;
  rand enum {HORIZONTAL, VERTICAL} direction;

  function new;
    if(pt1==null) pt1 = new;
    if(pt2==null) pt2 = new;
  endfunction

  constraint c_valid {
    if(direction == VERTICAL) {
      pt1.x == pt2.x;
      pt1.y != pt2.y;
    }

    if(direction == HORIZONTAL) {
      pt1.y == pt2.y;
      pt1.x != pt2.x;
    }
  }
endclass
```

Figure 5. Hierarchical (global) constraints

2.6 Reusable Constraints: Policy Classes

It is sometimes desirable to reuse constraints in several different objects, or to add additional constraints to an object at runtime. The technique described in [3] allows constraints to be defined in a container ("policy class") which can be added to another object at runtime.

These policy classes are derivatives of a parameterized base class (rand_policy_base). This base class defines a single member (item) of the same type as the class into which the constraints will be added. Policy constraints are written using the "item" handle to access the fields of the top-level object being randomized. The top-level object declares a rand queue of rand_policy_base objects. The top-level object's pre_randomize function iterates through the added policy instances and sets their "item" to point to the top-level object.

We can use this technique to add additional constraints to the line-drawing example from the previous section. Figure 6 shows the policy base class and two derived policy classes:

line_direction_policy changes the distribution of horizontal vs. vertical lines.

line_range_policy overrides the range of valid values for the line's endpoints. Because the XY point class already constrains the range of valid values, we must first disable that constraint using the constraint_mode function. We do this in the policy class pre_randomize function.

```
class rand_policy_base #(type T=uvm_object);
    T item;

    virtual function void set_item(T item);
        this.item = item;
    endfunction
endclass

class line_direction_policy extends rand_policy_base#(Line);
  constraint c_direction {
    item.direction dist {Line::VERTICAL := 5, Line::HORIZONTAL := 1};
  }
endclass

class line_range_policy extends rand_policy_base#(Line);
  function void pre_randomize;
    item.pt1.c_range.constraint_mode(0);
    item.pt2.c_range.constraint_mode(0);
  endfunction

  constraint c_range_override {
    item.pt1.x inside {[0:1000]};
    item.pt1.y inside {[0:1000]};
    item.pt2.x inside {[0:1000]};
    item.pt2.y inside {[0:1000]};
  }
endclass
```

Figure 6. Policy base class and derived policies

Figure 7 shows how these policies are added to the Line class. The class is the same as before except for the addition of the pcy queue and the pre_randomize function. These policies are added to the Line object's policy queue and are randomized at the same time as the Line.

```
class Line;
  rand rand_policy_base#(Line)     pcy[$];

  function void pre_randomize;
    foreach(pcy[i]) pcy[i].set_item(this);
  endfunction
  ...
endclass

...

Line l = new;
line_range_policy     range_pcy = new;   // construct policies
line_direction_policy direction_pcy = new;

l.pcy = {range_pcy, direction_pcy};      // add policies to object
l.randomize;
```

Figure 7. Adding constraints to an object using a policy class

3. Real-World Examples

Now that we have some new tools in our constraint toolbox, let's apply them to a few typical real-world verification problems.

3.1 Example 1: Allocate Transactions to Masters and Slaves in Bus Fabric Testbench

One application of constraints is configuring the number of transactions to be issued in a multi-master/multi-slave environment. Figure 8 shows a typical testbench for a bus interconnect DUT. The DUT has a number of master ports, each driven by a bus master agent, and a number of slave ports each connected to a slave agent. Each master may target a particular slave – typically by using a transaction address which falls within the slave's address range.

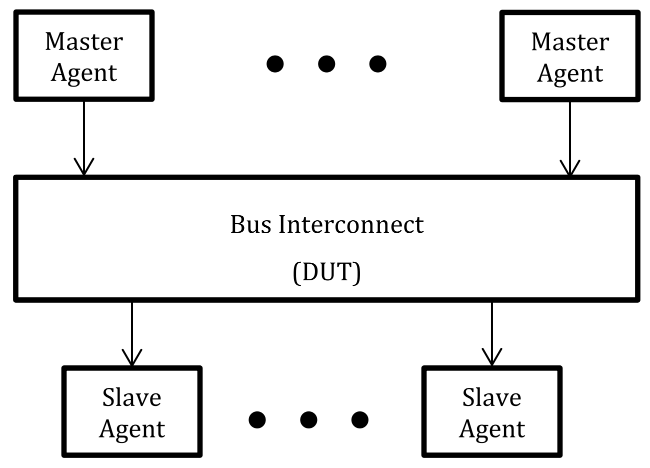

Figure 8. Typical bus fabric testbench

In the test environment, we want the ability to control the number of transactions generated in the test. We may want to control this in various ways:

- Total number of transactions for all masters
- Number of transactions per master.
- Number of transactions per slave
- Number of transactions from one master to a particular slave
- Avoid using certain masters or slaves (maybe that part of the DUT isn't working yet.)
- Use a particular master → slave path. (maybe there have been bugs in this area.)

We can control all of these with a constraint model which provides the necessary variables (or knobs) which can be constrained by the test writer.

The core of this model is a 2-dimensional array (txn_map) which indicates the number of transactions for each master/slave combination. The first array index indicates the master number; the second index indicates the slave.

```
rand int unsigned txn_map[N_MASTER][N_SLAVE];
```

Figure 9 shows an example txn_map array. The shaded portion represents the actual array. The far right column has the total number of transactions for each master (i.e. the sum of each row.) The bottom row has the total number of transactions for each slave (i.e. the sum of each column.)

	S0	S1	S2	S3	S4	S5	Total txns per master
M0	2	18	6	17	0	246	289
M1	1	0	3	128	0	155	287
M2	3	55	10	211	0	7	286
M3	15	0	97	2	0	175	286
M4	0	0	0	0	0	0	0
Total txns per slave	21	73	116	358	0	583	Total txns 1151

Figure 9. Example txn_map array

We can constrain the total number of transactions issued by all masters with an array reduction constraint which sums all of the entries in the txn_map array. The sum method used in this constraint sums all the elements over both array dimensions. We constrain this equal to a rand variable (total_txns) to provide a handy knob for subsequent test constraints.

```
rand int unsigned total_txns;
constraint c_valid {
  total_txns == txn_map.sum;
  // sum equivalent to: foreach(txn_map[m,s]) sum+=txn_map[m][s];
}
```

We can constrain the number of transactions for each master by summing the rows of the txn_map array. Because a SystemVerilog 2D array is essentially an array of arrays, we can use a sum-reduction constraint on each row. We constrain each row sum to equal the corresponding entry in the n_txns_per_master array, which gives us another set of control knobs for test constraints.

```
rand int unsigned n_txns_per_master[N_MASTER];
constraint c_valid {
   foreach(n_txns_per_master[m]) {
       n_txns_per_master[m] == txn_map[m].sum;
   }
 }
```

Constraining the number of transactions per slave requires summing the columns of the txn_map array. This requires a more elaborate constraint. We sum the entire array and use a "with" expression to limit the sum to those values where the array slave index matches the particular slave being summed. Within the "with" expression, we use the built-in "item.index" method to retrieve the 2nd array index (the slave index) for the current array element:

```
rand int unsigned n_txns_per_slave[N_SLAVE];
constraint c_valid {
   foreach(n_txns_per_slave[s]) {
      n_txns_per_slave[s] == txn_map.sum with(
         (item.index(2) == s) ? item : 0);
       );
      // sum is equivalent to:
      // foreach(txn_map[M,S])
      //       if(S == s) sum += txn_map[M][S];
      //       else       sum += 0;
     }
 }
```

We can constrain the use (or non-use) of a particular master or slave by testing if the number of transactions for that master or slave is greater than 0. Note that the two foreach constraints use the "if and only if" constraint style described in [4] (i.e. expr1 == expr2) instead of the "if..then" style which results from implication operator (expr1->expr2).

```
rand bit          use_master[N_MASTER];
rand bit          use_slave[N_MASTER];
constraint c_valid {
  foreach(use_master[m]) {
    (use_master[m] == 1) == (n_txns_per_master[m] > 0);
  }

  foreach(use_slave[s]) {
    (use_slave[s] == 1) == (n_txns_per_slave[s] > 0);
  }
 }
```

We can control the number of masters or slaves being used (i.e. they have a non-zero number of transactions) by summing the use_master and use_slave arrays. Because these are bit arrays and because the sum reduction function returns the same type as the array being summed (i.e. bit) we need to use a "with" expression to cast each array element to int before summing. This ensures the sum will not overflow:

```
rand int unsigned use_n_masters;
rand int unsigned use_n_slaves;

constraint c_valid {
   use_n_masters == use_master.sum with (int'(item));
   use_n_slaves  == use_slave.sum  with (int'(item))
}
```

To maximize interactions in the DUT, we may want to generate roughly the same number of transactions for each master. We constrain each master to have between min_val and max_val transactions. Those are two rand variables; all that we constrain is their difference. Their actual values will vary, based on the total number of transactions to be allocated.

```
rand int unsigned min_val;
rand int unsigned max_val;

constraint c_valid {
  foreach(n_txns_per_master[m]) {
    if(use_master[m])
          n_txns_per_master[m] inside {[min_val:max_val]};
  }
  soft (max_val - min_val) <= 10;
}
```

We can combine these variables and constraints into a master/slave configuration class (ms_config) as shown in Figure 10. This class can be randomized at the start of simulation to allocate transactions. The randomized config class can then be used to program or constrain the individual master agents to produce the specified number of transactions to each slave agent.

```
class ms_config#(int N_MASTER=3, int N_SLAVE=4);
  rand int unsigned txn_map[N_MASTER][N_SLAVE];
  rand int unsigned n_txns_per_master[N_MASTER];
  rand int unsigned n_txns_per_slave[N_SLAVE];
  rand int unsigned total_txns;
  rand int unsigned min_val;
  rand int unsigned max_val;
  rand bit          use_master[N_MASTER];
  rand bit          use_slave[N_MASTER];
  rand int unsigned use_n_masters;
  rand int unsigned use_n_slaves;

  constraint c_valid {
    total_txns == txn_map.sum;

    use_n_masters inside {[1:N_MASTER]};
    use_n_slaves  inside {[1:N_SLAVE]};

    foreach(txn_map[m,s]) txn_map[m][s] inside {[0:total_txns]};

    foreach(n_txns_per_master[m]){
      if(use_master[m])
                 n_txns_per_master[m] inside {[min_val:max_val]};
    }
    foreach(n_txns_per_master[m]) {
        n_txns_per_master[m] == txn_map[m].sum;
    }
    foreach(n_txns_per_slave[s]) {
      n_txns_per_slave[s] == txn_map.sum with(
                             (item.index(2) == s) ? item : 0);
    }
    foreach(use_master[m]) {
      (use_master[m] == 1) == (n_txns_per_master[m] > 0);
    }
    foreach(use_slave[s]) {
      (use_slave[s] == 1) == (n_txns_per_slave[s] > 0);
    }
    use_n_masters == use_master.sum with (int'(item));
    use_n_slaves  == use_slave.sum with (int'(item));
  }
  constraint reasonable_total_txns {
    soft total_txns inside {[0:9999]};
  }
  constraint reasonable val_range {
    soft (max_val - min_val) <= 10;
  }
endclass
```

Figure 10. Constraints for master/slave transaction allocation

Now that we have a full-featured constraint model, we can add additional constraints to it to tailor the stimulus for different tests. A test might use inline constraints to create a specific scenario as shown in Figure 11.

```
   ms_config cfg = new;
   cfg.randomize with {
     total_txns == 10000;           // Generate a total of 10000 txns
     use_n_masters inside {[2:4]};  // Use from 2-4 masters
     use_master[1] == 1;            // Use master 1
     use_slave[3] == 0;             // Don't use Slave 3
    }
```

Figure 11. Using inline constraints to control randomization of config class

Instead of creating inline constraints for each test scenario, we may want to control the configuration from the simulator command line. We can define plusargs for each of the control variables in the configuration. If the plusarg is specified on the simulator command line, we set the corresponding variable to the specified value and use rand_mode to disable the randomization of that variable. For array variables, we can dynamically build the format string for the $value$plusargs functions.

Figure 12 shows how this might be done for a few of the control variables. This particular scheme only allows pinning a variable to a single value. A more elaborate scheme would be needed to support specifying value ranges from the command line. UVM environments may choose to use the uvm_config_db and its command line options instead of $value$plusargs.

```
function void ms_config::get_plusargs;
  if($value$plusargs("total_txns=%d", total_txns))
     total_txns.rand_mode(0);

  if($value$plusargs("use_n_masters=%d", use_n_masters))
     use_n_masters.rand_mode(0);

  foreach(use_master[m]) begin
    string format = $sformatf("use_master[%0d]=%%b", m);
    if($value$plusargs(format, use_master[m]))
      use_master[m].rand_mode(0);
  end
  ...
endfunction


// Test config randomization
  ms_config cfg = new;
  cfg.get_plusargs;
  cfg.randomize;

// VCS simulation command line:
simv +total_txns=10000 +use_n_masters=3 '+use_master[1]=1' ...
```

Figure 12. Using command line plusargs to control randomization of config class

3.2 Example 2: Memory Map and Transaction Address Allocation

A common problem for memory system testing is the allocation of a memory map and generation of interesting transaction addresses using that map. A memory map will typically define a number of non-overlapping address ranges. Each address range may have different size and memory attributes (cacheable/noncacheable, instruction/data, etc.) We want to randomly create this map but allow the test to easily constrain its characteristics.

For this example, assume we want to generate a memory map for a hypothetical memory management unit with these characteristics:

- 32-bit address.
- 3 possible range (page) sizes: 1KB, 4KB, 1MB.
- Range starting addresses are aligned to the size of the range.
- Each range may be one of: non-cacheable, write-through cacheable or writeback cacheable.

We start our model by creating a class to represent a single memory range (mem_range), shown in Figure 13. The constraints in this class enforce the page size alignment requirement.

```
typedef bit [31:0]                         addr_t;
typedef enum addr_t { SIZE_1K='h400,
                      SIZE_4K='h1000,
                      SIZE_1M='h100000} size_t;

typedef enum        { NONCACHEABLE,
                      WT_CACHEABLE,
                      WB_CACHEABLE }    cache_t;

class mem_range;
  rand bit        valid;
  rand size_t     size;
  rand addr_t     start_addr;
  rand addr_t     end_addr;
  rand cache_t    cache_type;

  constraint c_valid {
    size == (1 + end_addr - start_addr);
    (start_addr & (size - 1)) == 0;    // align to page size
  }
endclass
```

Figure 13. Memory range class

The memory map class (mem_map) shown in Figure 14 contains a dynamic array of mem_range objects along with constraints to ensure ranges do not overlap. To simplify the constraints, we assume that ranges will be allocated sorted by address. Our constraint only needs to enforce that a range's starting address is greater than the previous range's ending address.

Before we can randomize the mem_map, we need to allocate the mem_ranges array and ensure each of its entries is non-null (SystemVerilog won't automatically call new for array elements). We do this in the pre_randomize function. The array is allocated to contain a user-specified maximum number of entries (max_ranges). This is an upper bound on the number of ranges that will be randomized.

The actual number of valid ranges is controlled by a rand variable (num_ranges) which is constrained by the valid bit in each mem_range. This trick lets us bypass the SystemVerilog restriction that an array size constraint is solved before array iterative constraints. In mem_map, we aren't changing the size of the array, but by changing the number of valid entries, we can get a similar end result. A key difference is num_ranges and the mem_range valid bits are all solved simultaneously, without any variable ordering. So, iterative constraints on the mem_ranges can affect the number of valid ranges. We also constrain the valid ranges so they are all grouped in the first num_ranges consecutive entries of the array.

```
class mem_map;
  int                            max_ranges = 10;
  rand int                       num_ranges;
  rand mem_range                 ranges[];
  rand rand_policy_base#(mem_map) pcy[$];

  function new(int max_ranges);
    this.max_ranges = max_ranges;
  endfunction

  function void pre_randomize;
    ranges = new[max_ranges];
    foreach (ranges[i]) ranges[i] = new();
    foreach(pcy[i]) pcy[i].set_item(this);
  endfunction

  constraint c_valid {
    num_ranges inside {[1:max_ranges]};
    foreach(ranges[i]) {
      ranges[i].valid == ((i < num_ranges) ? 1 : 0);
    }

    foreach(ranges[i])
      if(i>0) {
        if(ranges[i].valid)
          ranges[i].start_addr > ranges[i-1].end_addr;
      }
  }
endclass
```

Figure 14. Memory map class

The mem_map class has the hooks to support adding policy classes. Two example policies are shown in Figure 15. The first policy (page_count_policy) provides a way to constrain the number of pages of a particular size. It uses a sum-reduction constraint to iterate through the mem_range array looking for valid pages of a particular size. Note that the default reduction iterator variable "item" is overridden to avoid confusion with the "item" variable used in the policy class:

```
sum(r) with (r.valid ...)  vs. sum with (item.valid ...)
```

The second policy (cache_size_policy) enables constraining the total number of bytes of a particular cache type (not necessarily consecutive ranges). It also uses sum-reduction as in the first policy.

```
// Allocate between min and max pages of specified size
class page_count_policy extends rand_policy_base#(mem_map);
  addr_t size;
  int    min;   // minimum number of pages of given size
  int    max;   // maximum number of pages of given size

  constraint c_count {
    item.ranges.sum(r) with (int'(r.valid && (r.size == size)))
                inside {[min:max]};
  }

  function new(addr_t size, int min, int max=min);
    this.size = size;
    this.min  = min;
    this.max  = max;
  endfunction
endclass

// Allocate between min and max bytes of a specified cache type
class cache_size_policy extends rand_policy_base#(mem_map);
  cache_t cache_type;
  addr_t  min;
  addr_t  max;

  constraint c_cache {
    item.ranges.sum(r) with (
       (r.valid && (r.cache_type == cache_type))? r.size : 0
    ) inside {[min:max]};
  }

  function new(cache_t cache_type, addr_t min, addr_t max='1);
    this.cache_type = cache_type;
    this.min        = min;
    this.max        = max;
  endfunction
endclass
```

Figure 15. Memory range selection policies

Figure 16 shows how these policies are applied to mem_map randomization. We create multiple instances of each policy and use different constructor arguments to customize each policy's behavior. The use of policy classes allows us to create a complex constraint once, but reuse it multiple times with variations.

```
// Allocate memory map with:
// At most 2 4KB pages
// No 1MB pages
// At least 4KB of non-cacheable memory
// At least 8KB of write-back cacheable memory
// No write-though cacheable memory

mem_map            map = new;

page_count_policy count_4K_pcy = new(SIZE_4K, 0, 2);
page_count_policy count_1M_pcy = new(SIZE_1M, 0);

cache_size_policy NC_pcy = new(NONCACHEABLE, SIZE_4K);
cache_size_policy WB_pcy = new(WB_CACHEABLE, 2*SIZE_4K);
cache_size_policy WT_pcy = new(WT_CACHEABLE, 0,0);

map.pcy = { NC_pcy, WB_pcy, WT_pcy, count_4K_pcy, count_1M_pcy};

map.randomize;
```

Figure 16. Applying policies to mem_map randomization

Now that we have a configurable memory map, the final step in our example is to generate address transactions using the map. We want to generate transactions with these constraints:

- Transactions may be 1, 2 or 4 bytes.
- All bytes of a transaction must be within a single memory range
- Transaction addresses must be size-aligned
- Transaction's cache type must match its memory range cache type

Figure 17 shows how this is implemented. The transaction class (mem_txn) has constraints for size and alignment. It also has the usual hooks for adding policy classes. A policy class (txn_pick_policy) uses the pre-randomized mem_map to constrain the transactions address and cache type. Note in this case, the mem_map is not a rand variable, so its ranges will be used as state variables for the mem_txn randomization.

We use array reduction again, but this time, instead of sum-reduction, we use or-reduction. The constraint succeeds if the or returns a true value, which happens when the mem_txn attributes match (at least) one of the mem_ranges. Because the mem_ranges are non-overlapping, there can only be one range which matches the mem_txn.

```
class mem_txn;
  rand addr_t                    addr;
  rand addr_t                    size;
  rand cache_t                   cache_type;
  rand rand_policy_base#(mem_txn) pcy[$];

  constraint c_size  { size inside {1, 2, 4};}
  constraint c_align { (addr & (size-1)) == 0;}

  function void pre_randomize;
    foreach(pcy[i]) pcy[i].set_item(this);
  endfunction
endclass

class txn_pick_policy extends rand_policy_base#(mem_txn);
  mem_map  map;

  function new(mem_map map);
    this.map = map;

    $display("pcy new");  this.map.display;
  endfunction

  constraint c_pick {
    map.ranges.or(r) with (
      item.addr inside {[r.start_addr : (r.end_addr - item.size)]} &&
      item.cache_type == r.cache_type
    );
  }
endclass

...

txn_pick_policy pick_pcy = new(mem_map);
mem_txn         txn = new;

txn.pcy = {pick_pcy};
txn.randomize;
```

Figure 17. Generating transaction address from memory map

4. Constraint Debug

Constraints may not always give the desired results. A variable might never get an expected value. Or the distribution of values might not be what is expected. Or, worse, the constraints may be inconsistent and the solver will fail.

For inconsistent constraints failures, VCS automatically displays the constraints and variables causing the problem. The constraints may be reformatted slightly. One particularly helpful thing

VCS does is to add parentheses around all expressions and sub-expressions to clearly show which operands are associated with which operators. The class in Figure 18 produces a solver failure when it is randomized. The intent of the constraint was to have x=2 if y=1 otherwise x=0.

```
class C;
  rand int x;
       bit y = 1;

  constraint c1 {
    x inside {0, 2};
    x == y ? 2 : 0;
  }
endclass
```

Figure 18. Class with constraint failure

The solver output in Figure 19 shows why this failed. Note the extra parentheses around the x == y in the constraint. The == operator has a higher priority than the ?: operator, so x is compared with y and the result of that equality test is used as the condition for the ?: operator. The fix for this is to add parentheses to the original constraint:

```
x == (y ? 2 : 0);
```

```
Solver failed when solving following set of constraints

bit[0:0] y = 1'h1;
rand integer x; // rand_mode = ON

constraint c1    // (from this) (constraint_mode = ON) (test.sv:7)
{
   (x inside {0, 2});
   ((x == y) ? 2 : 0);
}
```

Figure 19. VCS output for inconsistent constraints failure

If the solver doesn't fail but the constraints still give unexpected results, we need to use other debug methods. A low-tech way to debug the case where the solver never chooses an expected value for a rand variable is to is to add another constraint which explicitly constrains that variable to equal the desired value. (e.g. x == 27;) If the desired value is not possible due to other constraints, the solver will fail and display the conflicting constraints.

If forcing the desired value didn't cause a failure, then it's possible to enable debug output for the randomize call. Debug can be enabled for all randomize calls or targeted to a specific call. To enable debug for all randomize calls, add these options to the simv command:

```
simv +ntb_solver_debug=trace +ntb_solver_debug_filter=all
```

This will produce trace output similar to what is produced for a solver failure. To trace only a specific randomize call, you need to first determine the call serial number. Another simv option will display the serial numbers for each randomize:

```
   simv +ntb_solver_debug=serial
```

Once you know the serial number,

```
   simv +ntb_solver_debug=trace +ntb_solver_debug_filter=serial#
```

It's also possible to extract the randomize call into a standalone testcase. This can be useful for experimenting with the constraints without the overhead of the rest of the testbench:

```
   simv +ntb_solver_debug=extract +ntb_solver_debug_filter=serial#
```

Extracted tests will be written to the simv.cst/testcases directory.

The most powerful way to debug constraints is to use the interactive debug features in DVE or Verdi. A few of the interactive debug features are:

- Debug soft constraints (show which constraints are dropped)
- Interactively modify constraints
- Re-randomize without recompiling

A full discussion of interactive constraint debug features is beyond the scope of this paper. See the DVE or Verdi documentation for more details.

5. Conclusions

This paper has reviewed several useful techniques using SystemVerilog constraints and showed how to combine them to tackle complex randomization problems. The use of array reduction constraints and the ability to encapsulate complex constraints into reusable policy classes were particularly useful in these examples. When constraints go wrong, VCS offers several ways, either command line or interactive, for debugging constraints.

6. References

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

[2] Accellera, "Universal Verification Methodology (UVM) 1.1 Class Reference" http://www.accellera.org/downloads/standards/uvm

[3] J. Dickol, "SystemVerilog Constraint Layering via Reusable Randomization Policy Classes," DVCon 2015. http://events.dvcon.org/2015/proceedings/papers/04P_11.pdf

[4] P. Marriott, J. Bromley, "Reverse Gear: Re-imagining Randomization using the VCS Constraint Solver", SNUG Silicon Valley 2014. https://www.synopsys.com/news/pubs/snug/2014/silicon-valley/mb06_marriott_paper.pdf