SystemVerilog Constraint Layering via Reusable Randomization Policy Classes

John Dickol
Samsung Austin R&D Center
Austin, TX
j.dickol@samsung.com

Abstract- SystemVerilog provides several mechanisms for layering constraints in an object. Constraints may be added via inheritance in a derived class. Inline constraints (i.e. randomize with {…} or `uvm_do_with) permit specifying additional constraints when randomizing an object. Unfortunately, SystemVerilog does not provide a good way to save these inline constraints for reuse in subsequent "randomize with" calls.

This paper describes a technique for packaging constraints into reusable "randomization policy" objects and using one or more of them to augment the constraints used while randomizing another object. Examples will be shown for both native SystemVerilog classes and UVM sequence items.

INTRODUCTION

SystemVerilog[1] classes and random constraints provide a powerful mechanism for creating verification stimulus. The language provides several mechanisms for layering constraints in a class object. Constraints may be added via inheritance in a derived class. Inline constraints (i.e. randomize with {…} or `uvm_do_with) permit specifying additional constraints when randomizing an object. Unfortunately, SystemVerilog does not provide a good way to save these inline constraints for reuse in subsequent "randomize with" calls.

This paper defines a way to encapsulate constraints into reusable "policy" classes and defines a methodology to use arbitrary combinations of these policies when randomizing another object. These policies may either be compiled into the other object or specified prior to randomization. Different test scenarios can use different combinations of policies as needed. This technique is applicable to native SystemVerilog or to a verification methodology such as UVM[2].

CONSTRAINT LAYERING SCENARIOS

Consider stimulus generation for a multi-processor memory system. Several types of constraints are typically needed for generating interesting transaction addresses:

- Limit addresses to the address ranges in the system memory map. These ranges may be dynamically generated at runtime and may change during the simulation.
- Avoid addresses in any reserved regions (e.g. "magic" addresses used for testbench control)
- Constrain addresses to cause evictions from the data cache. This requires keeping track of what addresses are currently in the cache and generating addresses which are not already in the cache.

In theory it's possible to include these types of constraints into a transaction base class with appropriate control knobs or add them as needed through inheritance. In practice, however, it is difficult to predict exactly what constraint capabilities a test writer will want to use. So, the test writer will add constraints in a new class derived from the base transaction. Different combinations of test constraints will require different derived classes.

Using inline constraints (randomize with{} or `uvm_do_with) is feasible for relatively simple constraints (e.g. number of transactions, simple distributions, etc.), but can be cumbersome for the more complex constraint types

listed above. These typically require elaborate iterative (foreach) constraints or require updating history information in the post_randomize method.

REVIEW OF EXISTING CONSTRAINT LAYERING TECHNIQUES

To demonstrate the existing and proposed constraint techniques, let's start by defining a simple transaction (addr_txn) consisting of an address and a size (number of bytes). From that base class we derive a read/write transaction (rw_txn):

```
class addr_txn;
  rand bit [31:0] addr;
  rand int        size;

 constraint c_size { size inside {1,2,4}; }
endclass
```

```
typedef enum {READ, WRITE} rw_t;

class rw_txn extends addr_txn;
 rand rw_t       op;
endclass
```

Figure 1. Address transaction base class and derived read/write transaction

We can constrain the address by adding constraints in a derived class (rw_constrained_txn):

```
class rw_constrained_txn extends rw_txn;
  constraint c_addr_valid {
    // Transaction addr range must fit within certain ranges
    addr inside {['h00000000 : 'h0000FFFF - size]} ||
    addr inside {['h10000000 : 'h1FFFFFFF - size]} ;

    // transaction must avoid "magic" testbench control addresses
    !(addr inside {['h13000000 : 'h130FFFFF - size]}) ;

    // Don't write to first 4K bytes.  Reads OK.
    if(op==WRITE) {
      !(addr inside {['h00000000 : 'h00000FFF - size]}) ;
    }
 }
endclass
```

Figure 2. Read/write transaction with address constraints in derived class

Or, we can specify inline constraints when randomizing the base object:

```
rw_txn t      = new;
...
t.randomize with {
  // Transaction addr range must fit within certain ranges
  addr inside {['h00000000 : 'h0000FFFF - size]} ||
  addr inside {['h10000000 : 'h1FFFFFFF - size]} ;

  // transaction must avoid "magic" testbench control addresses
  !(addr inside {['h13000000 : 'h130FFFFF - size]}) ;

  // Don't write to first 4K bytes.  Reads OK.
  if(op==WRITE) {
    !addr inside {['h00000000 : 'h00000FFF - size]} ;
  }
};
```

Figure 3. Randomizing read/write transaction using inline constraints

Although both of these techniques are usable, as more types or combinations of constraints are desired, it will be cumbersome to manage the many derived classes or sets of inline constraints. It would help if there were a way to package constraints into reusable building blocks.

HIERARCHICAL CONSTRAINT CONTAINERS

SystemVerilog provides a mechanism for hierarchical constraints. The language reference [1] defines "global constraints" but a better term for this might be "hierarchical constraint classes". A class may declare an object

member (i.e. class instance) as "rand". When the top-level object is randomized, the lower level objects are also randomized. All rand variables and constraints in the top- and lower-level objects are solved simultaneously.

We can apply this idea to our problem by moving sets of constraints into separate classes and declaring rand class handles for those constraint classes in the base transaction. Now, when the top-level transaction is randomized, the sub-class instances are randomized at the same time. We have problem however: The constraint classes have their own "addr" and "size" members which will be independently randomized. We need a way to ensure that these all have the same value. One way to solve this is adding additional "equality constraints" at the top level to ensure the addr and size members in each class have the same value. This works, but requires the top-level constraints to know which lower-level rand variables are in use. We will subsequently show a better way to solve this.

```
class addr_permit;
  rand bit [31:0] addr;
  rand int        size;
  constraint c_addr_permit {
  addr inside {['h00000000 : 'h0000FFFF - size]} ||
  addr inside {['h10000000 : 'h1FFFFFFF - size]} ;
  }
endclass

class addr_prohibit;
 rand bit [31:0] addr;
  rand int        size;
  constraint c_addr_prohibit {
   !(addr inside {['h13000000 : 'h130FFFFF - size]}) ;
  }
endclass
```

```
class rw_constrained_txn extends rw_txn;

   rand addr_permit permit     = new;
   rand addr_prohibit prohibit = new;

   // Ensure all addr & size are equal
   constraint c_all {
    this.addr == permit.addr;
    this.addr == prohibit.addr;

    this.size == permit.size;
    this.size == prohibit.size;
   }
endclass
```

Figure 4. Read/Write transaction with address constraints in separate container classes

We can more easily support multiple constraint classes in the top level by deriving all policies from a common base class and using a queue to contain any number of constraint classes. Using a foreach constraint to constrain the addr and size members will add the equality constraints for any number constraint containers.

```
class addr_constraint_base;
 rand bit [31:0] addr;
 rand int        size;
endclass

class addr_permit extends addr_constraint_base;
  constraint c_addr_permit {
   addr inside {['h00000000 : 'h0000FFFF - size]} ||
   addr inside {['h10000000 : 'h1FFFFFFF - size]} ;
 }
endclass

class addr_prohibit extends addr_constraint_base;
 constraint c_addr_prohibit {
  !(addr inside {['h13000000 : 'h130FFFFF - size]});
 }
endclass
```

```
class rw_constrained_txn extends rw_txn;
  rand addr_constraint_base cnst[$];

  function new;
    addr_permit permit     = new;
    addr_prohibit prohibit = new;
    cnst = {permit, prohibit};
 endfunction

 constraint c_all {
  foreach(cnst[i]) {
    this.addr == cnst[i].addr;
    this.size == cnst[i].size;
  }
 }
endclass
```

Figure 5. Read/Write transaction using a queue of constraint container classes

It can be a maintenance chore to keep the top-level equality constraints in sync with the constraint objects. For example, if we want to add a new constraint using rw_txn's "op" member (READ or WRITE), we need to update the top-level constraints to support this new member. If a constraint class doesn't constrain one of the top-level members, it still needs to declare it for the top-level equality constraints to be valid.

ELIMINATING TOP-LEVEL EQUALITY CONSTRAINTS

If the top-down equality constraints are problematic, perhaps can we use bottoms-up constraints instead? If the constraint objects had a way to directly refer to members of the containing class instance, we would not need to maintain the top-level equality constraints. SystemVerilog supports "upwards name referencing" which tries to resolve variable names by scanning upwards through the *module* hierarchy (see [1] section 23.8). This is close to what we want, except we want to scan upwards through the *class instance* hierarchy which is not supported.

What we can do, however, is declare an object handle in each constraint class which will contain a reference to the top-level object being randomized. If we write our constraints using this handle, we have access to all of the top-level objects members with requiring any top-level equality constraints. Figure 6 shows an example of this technique. The constraint base class (addr_constraint_base) contains a variable "item" of the same type as the top-level object (addr_txn). The constraints in each constraint class refer to the top level members by using the item handle (item.addr, item.size, etc.)

One step remains for this technique to work: we need to set the "item" variable to point to the top-level object before randomizing. This is conveniently done in the pre_randomize method of the top-level object.

```
class addr_constraint_base;
 addr_txn item;
endclass

class addr_permit extends addr_constraint_base;
 constraint c_addr_permit {
   // Transaction addr range must fit within certain ranges
   item.addr inside {['h00000000 : 'h0000FFFF - item.size]} ||
   item.addr inside {['h10000000 : 'h1FFFFFFF - item.size]} ;
 }
endclass

class addr_prohibit extends addr_constraint_base;
  constraint c_addr_prohibit {
    !(item.addr inside {['h13000000 : 'h130FFFFF - item.size]}) ;
  }
endclass

class rw_constrained_txn extends rw_txn;
 rand addr_constraint_base cnst[$];

 function new;
   addr_permit permit     = new;
   addr_prohibit prohibit = new;
   cnst = {permit, prohibit};
 endfunction

 function void pre_randomize;
    // Set item variable in each constraint class to point to top-level object being randomized
   foreach(cnst[i]) cnst[i].item = this;
 endfunction

endclass
```

Figure 6. Constraint class using item handle instead of top-level equality constraints

RANDOMIZATION POLICY CLASSES

In our examples, the constraint class has a hard-coded item class type. This can be made more generic by defining a parameterized base class with a type parameter indicating the type of the top-level object being randomized. We also add a function set_item for setting the item handle. We'll call this new type of container a "randomization policy".

```
class policy_base#(type ITEM=uvm_object);
  ITEM item;

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

endclass

class addr_txn;
 rand bit [31:0]             addr;
 rand int                    size;
 rand policy_base#(addr_txn) policy[$];

 constraint c_size { size inside {1,2,4}; }

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

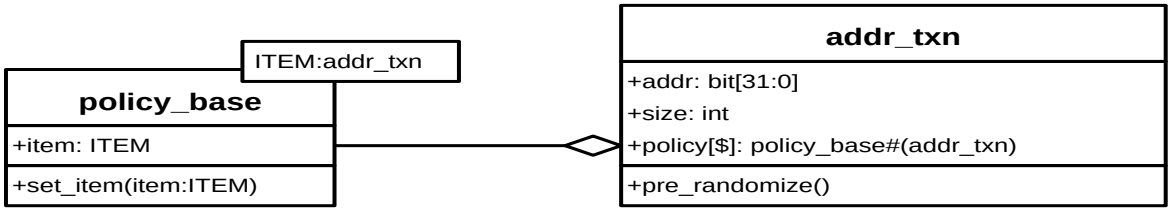

Figure 7. Randomization Policy base class example and UML class diagram

It would be handy to be able to bundle multiple policies into a single object. We can do this by creating a policy_list class which contains a queue of policies. The set_item method is overridden to set the item handle for all policies in the list. This allows recursively setting the item handle with a single call to the top-level set_item. With this technique, we can group interesting policies together and pass them around with a single assignment.

```
class policy_list#(type ITEM=uvm_object) extends policy_base #(ITEM);
  rand policy_base#(ITEM) policy[$];

  function void add(policy_base#(ITEM) pcy);
    policy.push_back(pcy);
  endfunction

  function void set_item(ITEM item);
    foreach(policy[i]) policy[i].set_item(item);
  endfunction
endclass

class rw_constrained_txn extends rw_txn;
  function new;
    addr_permit permit     = new;
    addr_prohibit prohibit = new;
    policy_list#(addr_txn) pcy = new;

    pcy.add(permit);
    pcy.add(prohibit);
    policy = {pcy};
  endfunction
endclass
```

Figure 8. Bundling multiple policies into a policy_list object

These policy lists can be nested to any number of levels. For example we may want to bundle the default address map policies (i.e. permitted and prohibited addresses) into a single default policy object and subsequently add that to a higher level policy object.

```
class cache_evict extends policy_base#(addr_txn);
  // constraints which cause cache evictions
endclass

class rw_constrained_txn extends rw_txn;
 function new;
   addr_permit   permit   = new;
   addr_prohibit prohibit = new;
   cache_evict   evict    = new;

   policy_list#(addr_txn) default_pcy = new;
   policy_list#(addr_txn) test_pcy    = new;

   default_pcy.add(permit);
   default_pcy.add(prohibit);

   test_pcy.add(default_pcy);
   test_pcy.add(evict);

   this.policy = {test_pcy};
 endfunction
endclass
```

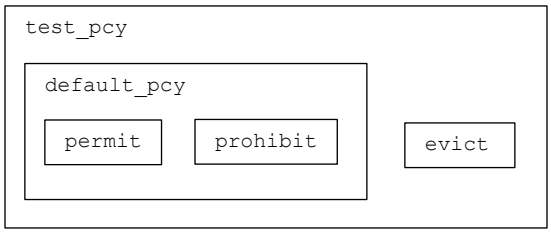

Figure 9. Nested policy classes

APPLICATIONS OF THE IDEA

Now that we have a convenient way to package constraints, we can create a few generic constraint policies. The address range examples used so far have had hard-coded address ranges. We can create configurable permit/prohibit address policies:

```
class addr_policy_base extends policy_base#(addr_txn);
  addr_range  ranges[$];

  function add(addr_t min, addr_t max);
    addr_range rng = new(min, max);
    ranges.push_back(rng);
  endfunction
endclass

class addr_permit_policy extends addr_policy_base;
 rand int selection;

  constraint c_addr_permit {
   selection inside {[0 : ranges.size() - 1]};

   foreach(ranges[i]) {
    if(selection == i) {
     item.addr inside {[ranges[i].min: ranges[i].max - item.size]};
    }
   }
 }
endclass

class addr_prohibit_policy extends addr_policy_base;
  constraint c_addr_prohibit {
   foreach(ranges[i]) {
    !(item.addr inside {[ranges[i].min : 1 + ranges[i].max - item.size]}) ;
   }
 }
endclass

class rw_constrained_txn extends rw_txn;
  function new;
    addr_permit_policy     permit   = new;
    addr_prohibit_policy   prohibit = new;
    policy_list#(addr_txn) pcy = new;

    permit.add('h00000000, 'h0000FFFF);
    permit.add('h10000000, 'h1FFFFFFF);
    pcy.add(permit);

    prohibit.add('h13000000, 'h130FFFFF);
    pcy.add(prohibit);

    this.policy = {pcy};
  endfunction
endclass
```

Figure 10. Configurable address permit/prohibit policies

Some policies may require the use of persistent state information. One example is generating a series of transactions which cause cache eviction. In an N-way cache, accessing more than N addresses with certain address bits the same (known as the index bits) but other address bits (the tag bits) different will cause one of the N entries (cache lines) already in the cache to be evicted to make room for a new entry. We can create a cache_evict policy class to generate this series of addresses. We will need to keep track of which cache line addresses have been used. We can keep this state data with the policy class by adding a state variable – an array (queue) containing the last N used addresses. This can be updated in the policy object's post_randomize function – i.e. after the top-level object has been randomized, the policy class's post_randomize records whatever information is needed for future randomizations. In this simple example, the state info is kept in the policy object, but it could also reference some global state info maintained elsewhere in the environment.

```
class cache_evict_policy extends addr_policy_base;

   addr_t line_hist[$];
   int    index;

   function new;
     super.new;
     std::randomize(index) with { index inside {[0:'h3f]}; };
   endfunction

   constraint c_evict {
                       !((item.addr & 'hFFFFF000) inside {this.line_hist}); // different tag
                         (item.addr & 'h00000FC0) == (this.index << 6);     // same index
                      }

  function void post_randomize;
     line_hist.push_back(item.addr & 'hFFFFF000);
  endfunction
endclass
```

Figure 11. Cache evict policy using state variables set in post_randomize

APPLYING TO UVM

The techniques describe so far have not used any specific methodology but can be easily applied to a UVM environment. We only need to derive our existing transaction class from uvm_sequence_item and set the randomization policies in the uvm_sequence before randomizing the sequence_item. Instead of the one-step `uvm_do macro, we split this into three steps: `uvm_create, set the policies, and `uvm_rand_send.

```
  class addr_txn extends uvm_sequence_item;

  class my_seq extends uvm_sequence #(addr_txn);
   addr_permit_policy permit     = new;
   addr_prohibit_policy prohibit = new;

   task body;
     //`uvm_do(req);

     `uvm_create(req);
     req.policy = {permit, prohibit};
     `uvm_rand_send(req);
   endtask
endclass
```

Figure 12. UVM sequence which sets policy before randomization

To simplify this process, we might create a new macro: "uvm_do_with_policy" which combines these three steps and allows passing the policy objects as an additional macro argument.

```
  `define uvm_do_with_policy(SEQ_ITEM, CONSTRAINTS="{}", POLICY="{}")\
   `uvm_create(SEQ_ITEM)\
   SEQ_ITEM.policy = POLICY;\
   `uvm_rand_send_with(SEQ_ITEM, CONSTRAINTS)

  class my_seq extends uvm_sequence #(addr_txn);
   addr_permit_policy permit     = new;
   addr_prohibit_policy prohibit = new;

   task body;
     `uvm_do_with_policy(req, {}, {permit, prohibit} );
   endtask
endclass
```

Figure 13. UVM macro which sets policy before randomization

Another option is to use the UVM configuration database. This is handy if we want the policies applied in a top-level sequence to apply to any sequence items launched by any child sequences launched by the top level sequence. The top-level sequence/vsequence pokes a default policy object into the config_db and the sequence item's

pre_randomize method gets the default policy from the config_db – but only if the object doesn't already have a policy. This scheme allows the convenience of setting the policy once at the possible expense of additional config_db accesses. If necessary, performance may be improved by explicitly setting a sequence_item's policy before randomization as in the previous examples.

When writing to the config_db, we use the top sequence's full hierarchical name with a ".*" wildcard appended. When reading from the config_db, we use the sequence_item's full hierarchical name. This scheme lets us set a default policy for all items in a sequence hierarchy but allows overriding the default by adding additional config_db entries with a more specific path name.

```
class my_seq extends uvm_sequence #(addr_txn);
 ...
 policy_list#(addr_txn) default_pcy = new;
 policy_list#(addr_txn) special_pcy = new;

 task body;
  default_pcy.add(permit);
  default_pcy.add(prohibit);

  special_pcy.add(very_special_pcy);   // non-default policy for some sub sequences

  // write default policy into config_db using top sequence full name + wildcard.
  uvm_config_db#(policy_list#(addr_txn))::set(null,
                                              {get_full_name, ".*"},
                                              "default_policy",
                                              default_pcy);

 // Use special policy for items in the sub_seq.seq2 sequence (and below)
 uvm_config_db#(policy_list#(addr_txn))::set(null,
                                              {get_full_name, "sub_seq.sub2.*"},
                                              "default_policy",
                                              special_pcy);

  // default_pcy will be used by all sequences items started by following uvm_do calls
  // Except sub_seq.seq2 which will use special_pcy.
  `uvm_do(req);
  `uvm_do(sub_seq);
 endtask

class addr_txn extends uvm_sequence_item;
  ...
  function void pre_randomize;
   super.pre_randomize();

   // If policy queue is empty, attempt to get default policy from the config db.
   // Use item's fullname to query the config_db
   if(policy.size ==0) begin
     policy_list#(addr_txn) default_pcy;

     if(uvm_config_db#(policy_list#(addr_txn))::get(null,
                                                    get_full_name,
                                                    "default_policy",
                                                    default_pcy)     ) begin
      policy = { default_pcy };
     end else begin
       `uvm_error(get_type_name(), "could not get policy from config_db");
      end
   end

  foreach(policy[i]) policy[i].set_item(this);
endfunction
endclass
```

Figure 14. Using UVM config_db to get/set randomization policies

RESULTS

Performance of the constraint techniques described in this paper was measured using a simple UVM environment which randomized a transaction 10,000 times for each technique. Total runtime was measured from within the testbench by using DPI to read the time-of-day clock before and after the randomization sequence. Data was collected for two of the "big-three" SystemVerilog simulators.

Figure 15 shows the relative runtime of each technique as a percentage of the case where all constraints are in the base or derived class. We can make the following observations from this data:

- Inline constraints (randomize with{}) have virtually the same runtime as constraints specified in the class.
- Constraints in a policy class with "equality constraints" have the slowest runtime. This is likely due to the larger number of constraints to be solved.
- Constraints in a policy class using an "item handle" have only a modest (3-9%) runtime penalty provided the policy is explicitly set in the randomized transaction and not obtained from the config_db.
- Getting the constraint policy from the config_db increases runtime by ~16% vs. explicitly setting the policy before randomization.

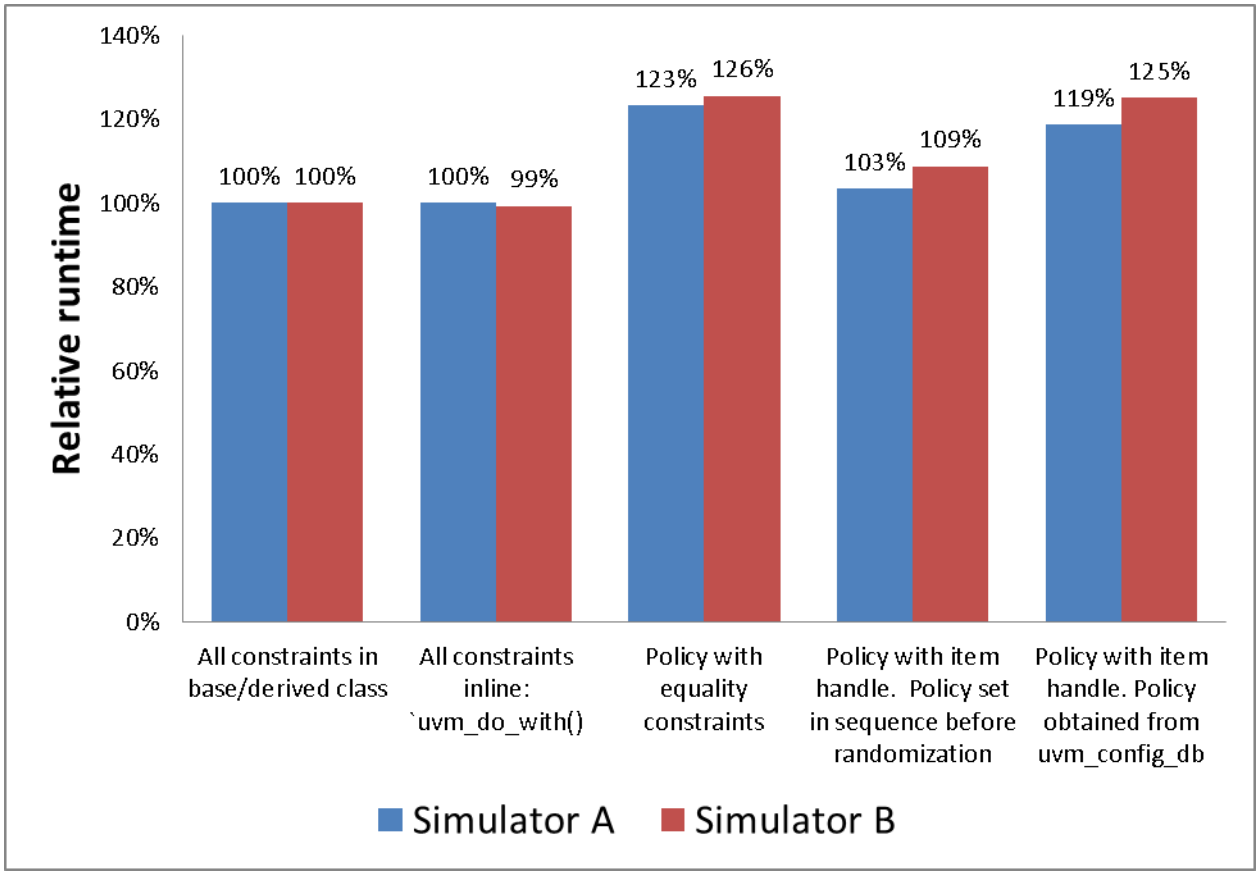

Figure 15.Relative performance of constraint techniques.

CONCLUSIONS

This paper shows a flexible technique for SystemVerilog constraint layering which offers comparable performance and improved ease-of-use over the existing approaches. The use of randomization policy classes provides a convenient and efficient way to mix and match different types of constraints into an object being randomized. This technique can be use with "raw" SystemVerilog or can be applied to UVM.

ACKNOWLEDGMENTS

The author would like to thank John Rose from Cadence for reviewing and sanity checking the initial ideas in this paper and for suggesting the "item" handle used in the policy classes.

REFERENCES

[1] *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2012
[2] UVM Documentation http://www.accellera.org/downloads/standards/uvm