SystemVerilog Constraint Layering via Reusable Randomization Policy Classes

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

Problem: How to reuse random constraints?

SystemVerilog constraints may be added to an object via inheritance or with inline constraints specified when the object is randomized (e.g. `obj.randomize with {…}`) But the SV language doesn't define a way to easily reuse constraints in multiple objects.

Solution: Put constraints in "Policy Classes"

Putting the constraints in a standalone class allows them to be defined once then added into other objects as needed. Policies can be mixed and matched in any combination.

Examples

The two examples below illustrate the concept for a simple address transaction. Two policies constrain the generated addresses to lie within permitted regions and outside prohibited regions.

More details in the paper

See the paper for more applications of this idea:

- policy_list classes encapsulate a list of policies. Lists may be nested to any number of levels.
- Policies with persistent state information e.g. keep track of recently used addresses and use them in constraints for subsequent randomizations.

Conclusions

Randomization policy classes provide a flexible and efficient way to add different types of constraints into an object being randomized. This technique can be used with native SystemVerilog or can be applied to UVM.

SV Example

This example uses policy classes to add additional reusable constraints to a transaction object.

Policy Base Class

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

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

Policy Classes

```
class addr_permit_policy extends policy_base#(addr_txn);

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

```
class addr_prohibit_policy extends policy_base#(addr_txn);

 // Transaction addr range must avoid prohibited ranges
 constraint c_addr_permit {
  !(item.addr inside {['h13000000 : 'h130FFFFF - item.size]});
 }
endclass
```

Transaction Class

```
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
```

Test Fragment

```
...
addr_permit_policy   permit   = new;
addr_prohibit_policy prohibit = new;
addr_txn txn = new;

txn.policy = {permit, prohibit};

txn.randomize;
...
```

1. Policy classes contain reusable constraints
2. "item" refers to object being randomized
3. "item" constraints apply to top-level object
4. Queue of policies for this transaction
5. Add policies to txn policy queue
6. Randomizing txn…
7. …sets "item" in each policy to point to this txn
8. Policies are randomized at same time as txn

UVM Example

This example adapts the SV example to UVM. A sequence adds the same policy classes to the same address transaction which has been converted to a UVM sequence item.

UVM Sequence

```
class my_seq extends uvm_sequence #(addr_txn);
 ...
 my_subsequence          sub_seq;
 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(default_pcy);
  special_pcy.add(special);

  // `uvm_do(req);
  `uvm_create(req);
  req.policy = {default_pcy};
  `uvm_rand_send(req);

  uvm_config_db#(policy_list#(addr_txn))::set(
                                null, {get_full_name, ".sub_seq.*"},
                                "default_policy", special_pcy);

  `uvm_do(sub_seq);

 endtask
```

UVM Transaction Class

```
class addr_txn extends uvm_sequence_item;
 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;
   super.pre_randomize();

   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
```

1. policy_list combines multiple policies into a single policy
2. Sequence sets default policy for sequence item "req"
3. Sequence puts special policy into config_db …
4. … using sub_seq fullname + wildcard …
5. … for use by sub_seq or any of its children
6. If policy has not already been set …
7. … try to get policy from UVM config_db
8. Use item's full path to query config_db
9. If successful, set policy for this sequence item