Polymorphic Transaction-Level Constraint

Thinh Ngo, Ph.D. Can Tho University, Vietnam

***Abstract*—Constrained randomization is performed in transactions (i.e., sequence_item) at a fine-grained signal-level control and in sequences at a course-grained control (e.g., rand_mode, constraint_mode). As a result, most randomization engineering and coding are centralized within a transaction type and operated at runtime with one-size-fits-all complexity that is susceptible to error and undermining reuse and scalability. This study presents Polymorphic Transaction-Level Constraint, where constrained randomization is additionally layered and distributed at a higher abstraction level. Specifically, polymorphism is applied to allow constrained randomization at the transaction type level. Instead of one type, multiple transaction types collaborate at runtime in the constrained randomization mechanics, allowing constraint encapsulation for better reuse and extensibility.**

I. Polymorphic Transaction-Level Constraint

Traditionally, multiple sequences with constraints of course controls (i.e., knobs) control fine-grained constrained randomization of transaction data. All data constraints are implemented and contained in a single transaction type. For small and simple interfaces, this centralized constraint approach is practical. However, the distributed approach would be a better alternative for large and complex interfaces (e.g., with various modes, data/control signal groups, and en/decoding operations). Polymorphic transaction-level constraint utilizes multiple transaction types to enable shared and distributed constraints at runtime. In effect, a layer of inter-transaction constraint is added between the sequence layer and the intra-transaction layer constraint.

II. Implementation

From a bottom-up perspective, multiple transaction types are derived from the base transaction type. These transaction types differ at higher signal abstraction levels, such as signal modes/representations. These transaction types are created and randomized before being randomly selected for downstream transmission. OOP polymorphism is the critical programming enabler to allow multiple objects related by inheritance to act as the base object throughout the testbench. From the top-down perspective, a sequence uses SystemVerilog constraints to randomize the selected transaction type. An additional class is created inside a sequence to perform the class-based constrained randomization of the transaction type enumeration. A workaround is performed to enable polymorphism in a UVM testbench, where items are instantiated using a new function instead of the factory create function. It would affect the transaction factory only. Codes are verified at EDA Playground [1] and archived at [2]. Additionally, the sequence, the transaction, and the driver codes are included below.

III. Conclusion

This research presents an implementation of Transaction-Level Constraint (TLC) to facilitate polymorphic constraint engineering and enhance the traditional constrained randomization methodology.

IV. Coding

```
class my_sequence extends uvm_sequence
        #(my_item);
 `uvm_object_utils(my_sequence)

 typedef enum { SMALL, ZERO, BIG}
        t_itemtype;
 typedef randomizetype;
 t_itemtype m_itemtype;
 my_itemSD reqSD;
 my_itemBD reqBD;
 my_itemZD reqZD;
 randomizetype m_randtype;

 function new(string name="");
  super.new(name);
  reqSD = new();
  reqBD = new();
  reqZD = new();
  m_randtype = new();
 endfunction

 virtual task body();
  repeat (50) begin
   if (!reqZD.randomize()) $dispaly("errZD");
   if (!reqSD.randomize()) $display("errSD");
   if (!reqBD.randomize()) $display("errBD");

   m_randtype.randomize();
   m_itemtype = m_randtype.gettype();
   case (m_itemtype)
    SMALL: req = reqSD;
    BIG:   req = reqBD;
    ZERO:  req = reqZD;
   endcase

   start_item(req);
   finish_item(req);
  end
 endtask

 //This class randomizes each transaction type
 class randomizetype;
  rand t_itemtype m_itemtype;

  //consraint of transaction types
  constraint c_itemtype {
   m_itemtype dist {SMALL:=1,ZERO:=2,BIG:=3};
  }

  function t_itemtype gettype();
   return m_itemtype;
  endfunction
 endclass
endclass
```

```
class my_itemBD extends my_item;
 `uvm_object_utils(my_itemBD)

 constraint c_delay
        {delay > 3; delay <6;}

 function new(string name="");
  super.new(name);
 endfunction

 virtual function void
        do_copy(uvm_object rhs);
  my_item tx;
  $cast(tx, rhs);
  super.do_copy(rhs);
  reset = tx.reset;
  enable = tx.enable;
  d = tx.d;
  q = tx.q;
  delay = tx.delay;
 endfunction

 virtual function int m_get_delay();
  return this.delay;
 endfunction

 virtual function logic m_get_reset();
  return this.reset;
 endfunction

 virtual function logic m_get_enable();
  return this.enable;
 endfunction

 virtual function logic m_get_d();
  return this.d;
 endfunction
endclass

class my_driver extends uvm_driver
        #(my_item);
 `uvm_component_utils(my_driver)

 virtual my_if vif;

 function new(string name="my_driver",
        uvm_component parent);
  super.new(name, parent);
 endfunction

 virtual function void build_phase
        (uvm_phase phase);
  super.build_phase(phase);
  if (!uvm_config_db#(virtual my_if)::
  get(this, "", "vif", vif))
  `uvm_fatal("VIF", "can't get vif")
 endfunction

 virtual task run_phase
        (uvm_phase phase);
  super.run_phase(phase);
   forever begin
    @(posedge vif.clk);
    seq_item_port.get_next_item(req);
    req.m_print("DRIVER");
    repeat (req.m_get_delay())
         @(vif.clk);
    vif.reset = req.m_get_reset();
    vif.enable = req.m_get_enable();
    vif.d = req.m_get_d();
    seq_item_port.item_done();
   end
  endtask
endclass
```

REFERENCES

[1] https://www.edaplayground.com/x/KkvV
[2] https://github.com/thinhngo11/Transaction-Level-Constraint