2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

# Conquer the Graphics Legacy: Develop the Customized UVM VIP Embedded a Cycle-based C++ Reference Model

Roman Wang
Advanced Micro Devices, Inc.
Roman.Wang@amd.com

***Abstract*-Graphics verification usually adopts the cycle-based C++ simulation model (CSIM) to represent the functional behavior of a design under test (DUT). The legacy block-level testbench (TB) uses a Verilog bus functional model (BFM) to interact with the CSIM which takes the client behavior role. The bridge between BFM and CSIM is either the direct programming interface (DPI) or SystemC proxy. This approach works well for years, but it really has several issues too challenging to migrate the legacy testbench into the universal verification methodology (UVM) testbench. This paper will discuss a legacy cases study and then illustrate a practical approach for customized UVM verification IP (VIP) or UVM verification component (UVC) embedded in the CSIM. It has already been implemented for some new UVM VIPs and is feasible.**

***Keywords*- UVM, UVC, VIP, CSIM, BFM**

## I. INTRODUCTION

### *A. Motivation*

In the legacy verification testbench, the C++ simulation model (CSIM) is created to represent the DUT behavior so called the reference model. It is cycle based but not cycle-accurate. The verification engineer writes the constraint random C++ test interacting with the CSIM and run the C++ code to generate several vector files in a specific format. One vector is related to a specific bus interface. The testbench integrates several Verilog BFMs and attaches them on the DUT. The host BFM will read its vector and drive on the bus. The testbench central control mechanism can coordinate multiple host BFMs in a reasonable rule. For some client BFMs, the functional behavior depends on its CSIM. The interoperability between the BFM and CSIM can occur in different ways (see Section II). With the rapid adoption of UVM [1][2] year-by-year, a modular and reusable UVM testbench is required. The biggest challenge is how to migrate Verilog BFM into the modular interface UVC or UVM VIP. Pure Verilog BFM is fairly simple to be migrated as the paper [3] mentioned years ago. This paper will focus on the migration of the client Verilog BFM with CSIM into a customized interface UVC.

### *B. Paper Organization*

Section II attempts to dissect the legacy cases study and challenges. Section III presents the proposed customized UVM VIP solution. This paper concludes in Section IV by sharing how we benefit from this solution. Finally, we will summarize some future work in Section V.

## II. THE LEGACY STUDY

- *Verilog BFM interacting with C++ model through the DPI*

As Fig. 1 shows, the client Verilog BFM captures the buses of DUT output, assembles the signals into several interface structs and refresh them to the CSIM per cycle. At the end of every cycle, it assembles the CSIM output interface structs into proper signals and refresh them on the Verilog BFM outputs.

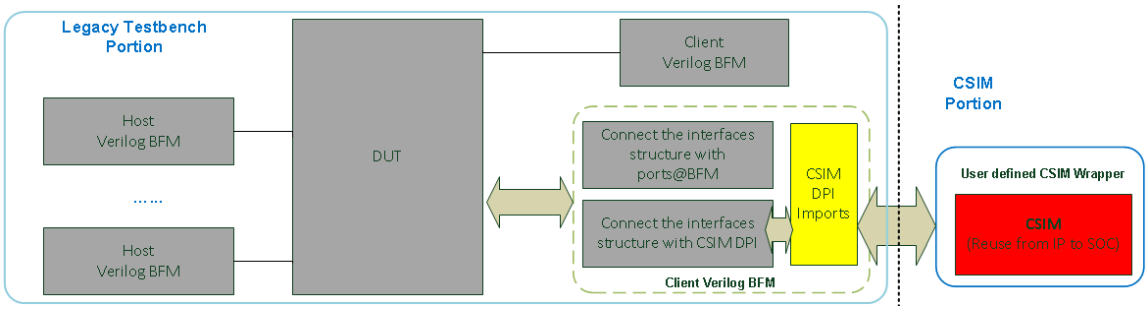


**Fig. 1. Verilog BFM interacting with C++ model through the DPI.**

- *Verilog TB tracker or monitor interacting with CSIM through Verilog PLI and SystemC proxy*

As Fig. 2 shows, the client Verilog BFM captures the buses of DUT output, assembles the signals together, broadcasts to a SystemC TB proxy by calling the PLI function per cycle. In parallel, it continues to poll the SystemC TB proxy to get the CSIM outputs by calling the PLI function per cycle, assembles the CSIM outputs into proper signals and refreshes the Verilog BFM outputs. On the CSIM portion, it also directly calls the PLI function to interact with the SystemC TB proxy.

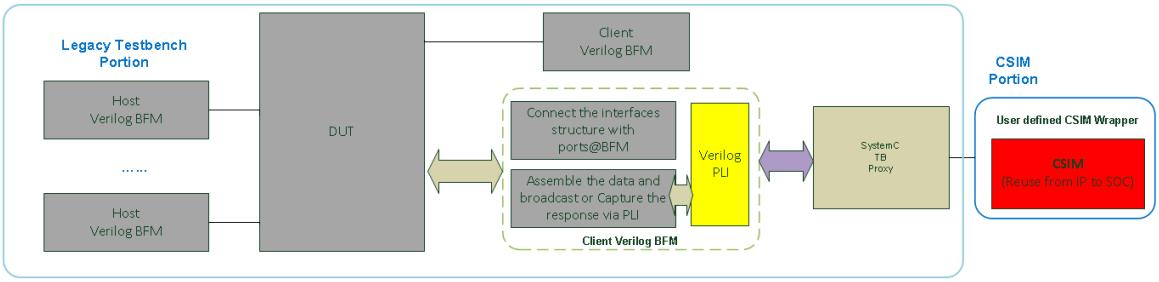


**Fig. 2. Verilog TB tracker or monitor interacting with CSIM through Verilog PLI and SystemC proxy.**

## III. PROPOSED SOLUTION

### *A. The concept of parameterized UVM vector agent*

When we review the bus interfaces on top level of design, they are several common features:

1. Most likely non-standard bus on protocol point of view.
2. Not complicated timing on each small bus.
3. A vector interface (with different width) catenated by specific bus signals.

To avoid creating multiple different interface UVCs per different buses, we create a generic parameterized UVM vector agent shown in Fig. 3. The high level UVC architecture looks simple and its mission is just to drive the parameter vector on the bus by either sync or async way. Its interface is parameterized for dut_in and dut_out definition. The parameterized interface is easily binding on design module ports and its virtual interface can be retrieved in UVC components by calling uvm_config_db. Its monitor can also spy the bus interface and broadcast the vector to high level monitor, the broadcast will happen either every cycle or if any bus signal changes per user configuration. The vector agent item has two sub-item objects: dut_ins and dut_outs. They are not meaningful to the verification engineer and must be translated into a meaningful transaction by using unpack in the UVM object automations. The monitor provides three UVM analysis ports: dut_in_ap, dut_out_ap and dut_in_out_ap. By default, the monitor will update both dut_ins and dut_outs, and it will broadcast it through dut_in_out_ap. In some cases, only dut inputs are active and outputs are idle. To improve the performance and make the TLM connection clear, the user can also select the active analysis port for high level integration (Ex. dut_in_ap).

Figure 4 shows an example of vector handle in high level external monitor. p_uvc_pkt #(10 , 20) means the vector agent item with 10 bis inputs and 20 bits outputs. When the user implements the do_unpack function in user_bus_item, the order is depending on the vector agent SV interface bind shown in Fig. 5.

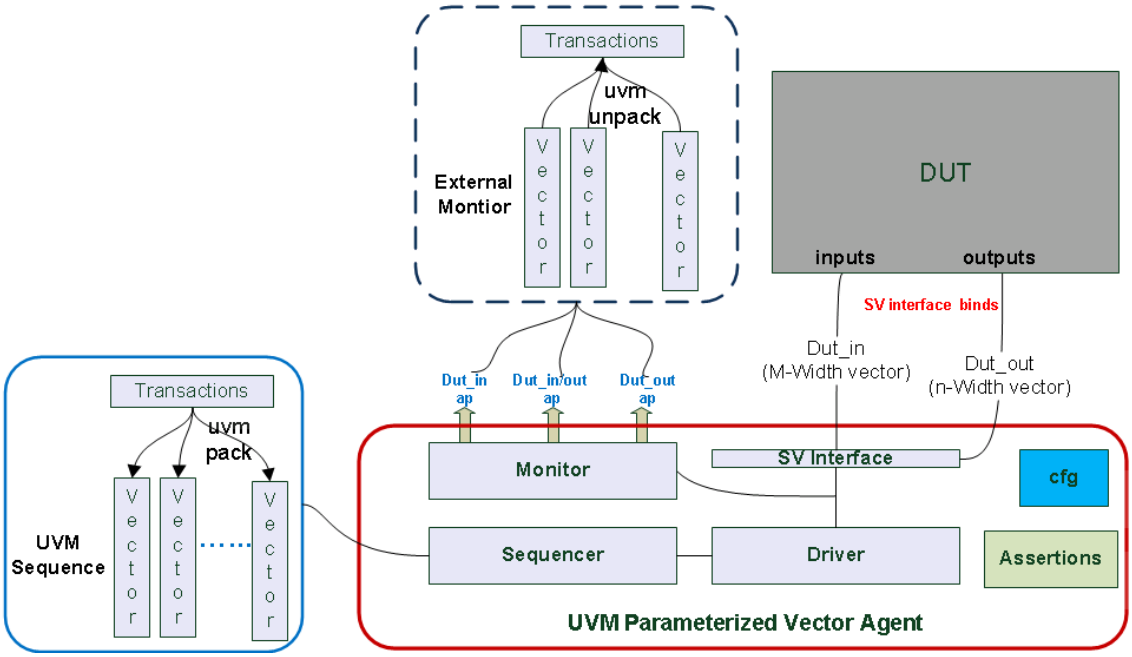


**Fig. 3. The concept of UVM parameterized vector agent.**

```
function void write_dut_outs_vector_imp_port(uvm_sequence_item t);

  p_uvc_pkt #(10 , 20)                dut_outs_pkt;
  user_bus_item                        bus_item;
  bit                                  m_bitstream[]   ; /// bit stream for unpack

  dut_outs_pkt = new ("dut_outs_pkt ");
  bus_item = user_bus_item::type_id::create("bus_item ",this);

  if (!$cast(dut_outs_pkt, t))
    `uvm_fatal(get_type_name(), " dut_outs_pkt type cast failed!")

  m_bitstream = new[$bits(dut_outs_pkt.dut_outs.value)];
  foreach(m_bitstream[i]) begin
    m_bitstream[i] = dut_outs_pkt.dut_outs.value[i];
  end
  bus_item.unpack(m_bitstream);
```

**Fig. 4. The UVM unpack code example.**

```
  bind design_module p_vector_if#( 10, 20)  p_vector_if_inst(
  .reset(~resetb),
  .clk(clk),
  .en(1'b1),
  .dut_inputs( {signal_in_1, signal_in_2, signal_in_3 }),      // vector_in = 3bits + 4bits + 3bits
  .dut_outputs({signal_out_1, signal_out_2, signal_out_3)  //  vector_out = 10bits+ 5bits + 5bits
  );
```

**Fig. 5. SV interface bind code example.**

*B. The customized UVM UVC VIP embedded a cycle-based C++ reference model*

The proposed UVC architecture is shown in Fig. 6, and it is implemented based on UVM parameterized vector agent. The goal is to reuse the existing CSIM without creating a new SystemVerilog (SV) reference model, because the other new created model will duplicate the effort and introduce more risk, like a new bug.

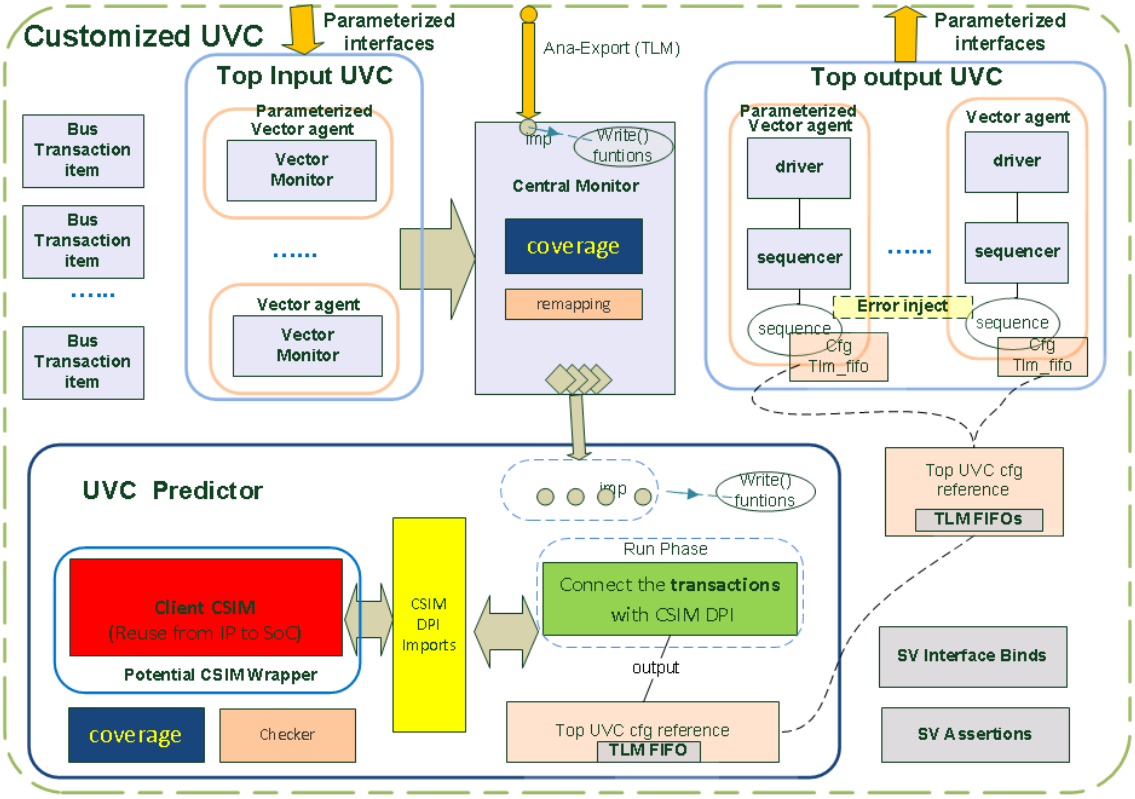


**Fig. 6. The architecture of customized UVC.**

The main portions of customized UVC are dissected as the basic dataflow:

- *Bus transaction item*

  We integrate multiple UVM parameterized vector agents to spy the input buses and drive output buses of customized UVC, but the vector bus item is not meaningful. We define the bus transactions to represent every bus for both input and output. In the input transaction item, it only implements the do_unpack function because of monitoring. In the output transaction item, it only implements the do_pack function because of driving. Figure 7 shows an implementation example of input bus transaction item.

- *Top input UVC*

  It is a collection of UVM parameterized vector agents, and the agent instance number is configurable per specific requirement. All of instances are passive, just spying each input bus interface and broadcasting the vectors to the central monitor.

- *Central monitor*

  The central monitor is working as a central subscriber. Its missions are including:

  - Collect all input vectors from top input UVC.
  - Translate the vector into a specific bus transaction item.

- Do potential re-remapping, checking, and built-in transaction based functional coverage. For example, it can create another transaction item by referring local transaction data.
- Broadcast the transaction items to the UVC predictor.
- UVM implementation port to receive external transaction by connection with TLM port. This may be used for remapping, checking, or functional coverage implementation.

```
class input_bus1_item extends uvm_sequence_item;

  bit [2:0] signal_in_1;
  bit [3:0] signal_in_2;
  bit [2:0] signal_in_3;

  `uvm_object_utils_begin(input_bus1_item)
      `uvm_field_int(signal_in_1, UVM_ALL_ON|UVM_NOCOMPARE|UVM_NOPACK)
      `uvm_field_int(signal_in_2, UVM_ALL_ON|UVM_NOCOMPARE|UVM_NOPACK)
      `uvm_field_int(signal_in_3, UVM_ALL_ON|UVM_NOCOMPARE|UVM_NOPACK)
   `uvm_object_utils_end

    function void do_unpack(uvm_packer packer);
      super.do_unpack(packer);
      this.signal_in_1    = packer.unpack_field_int($bits(signal_in_1));
      this.signal_in_2    = packer.unpack_field_int($bits(signal_in_2));
      this.signal_in_3    = packer.unpack_field_int($bits(signal_in_3));
    endfunction: do_unpack
```

**Fig. 7. The Transaction with pack implementation code example.**

❖ *UVC predictor*

It is a UVM env and act the reference model role in customized UVC. It receives the input transactions from central monitor, pushes them into specific tlm_fifo and broadcasts the CSIM output transactions to top output UVC. As Fig. 8 shows, we mainly implement the prediction by interworking with CSIM in the run_phase. There are two concurrent "forever begin/end" statement blocks.

The first thread is to process CSIM interoperability by calling proper DPI function:

- CSIM initialization.
- Try to get the transaction from tlm_fifo, assemble the valid transactions into an interface structs, and assign all zeros in case of invalid.
- Refresh the CSIM by feeding the interface structs.
- Retrieve the output interface structs from the CSIM.
- Assemble the CSIM output interface structs into the output transaction items and push them into the tlm_fifo inside the top cfg handle.
- Sample the CSIM output transaction based functional coverage.
- Delay one clock cycle.

The second thread is to achieve the reset aware operation.

- The mechanism is to wait the reset assert, then flush the CSIM and tlm_fifo inside top cfg handle.

❖ *Top output UVC*

It is a collection of UVM parameterized vector agents, and the agent instance number is configurable per specific requirement. All the instances are active, driving each output bus interface. There is a built-in free-running sequence on each vector agent. Its mission is to pop out the transaction from the tlm_fifo (inside the top cfg handle), pack it into a vector, and refresh on the output bus interface per cycle. If there is no existed transaction in the tlm_fifo, it will clear the bus. We can do the configurable error injection feature in the free running sequence.

- *SV interface binds*

  It is the UVM parameterized vector agent SV interfaces binding for both top input and output UVC.

- *SVA*

  We also implement the built-in assertions to check the critical timing on the bus interface between the DUT and UVC.

```
task run_phase(uvm_phase phase);
   uvm_sequence_item tr;
   phase.raise_objection(this);
   csim_init();   // initialize CSIM
   phase.drop_objection(this);

  fork
  forever begin
    cfg.wait_for_clock(1); // delay one clock cycle

    if(tlm_fifo1.try_get(tr)) begin
    $cast(item_in1, tr);
    if(item_in1.valid) begin
      input_if_struct1.valid= item_in1.valid;
      input_if_struct1.a      = item_in1.a;
      input_if_struct1.b      = item_in1.b[0];
    end
    else begin
      input_if_struct1.valid= 0;

      input_if_struct1.a      = 0;
      input_if_struct1.b      = 0;
    end

     drive_csim_input(); //drive CSIM inputs
     csim_run();
     track_emu_output(); // get CSIM outputs

     if(output_if_struct1.valid) begin
        out_item1 =out_item::type_id::create("out_item1",this);
        out_item1.valid = output_if_struct1.valid;
        out_item1.data = output_if_struct1.data;
        cfg. export_fifo.push_back(out_item1.clone());
     end

  // Handle inflight reset
  forever begin
    uvm_sequence_item item;
    cfg.wait_for_reset();
    csim_reset();
    while (cfg.export_fifo.size() != 0)
      _item = cfg.VM_export_fifo.pop_front();
  end
 join

endtask: run_phase
```

**Fig. 7. The UVC predictor code example.**

```
vector_pkt #(IN_W, OUT_W )  pkt;
out_seq_item    tr;
bit bits_stream[];
bit [UVM_STREAMBITS-1:0] vector;

virtual task body();
  super.body();
  forever begin
    if (m_cfg.export_fifo.size() > 0) begin
      tr = m_cfg. export_fifo.pop_front();
      tr.pack(bits_stream);
      foreach(bits_stream[i])
      vector[i]=bits_stream[i];
      `uvm_do_with (pkt, { pkt.dut_ins.value == vector;})
      get_response(rsp); // just drop here
   end
endtask: body
```

**Fig. 8. The free-running sequence code example.**

## C. *Adoption examples*

When the customized UVC is ready, there are two adoption choices for user per specific requirement.

- Integrate the customized UVC in the legacy test bench

  Figure 9 shows the overview testbench architecture with UVC integration. The code example is shown in Fig. 10. We instance the UVC and its configuration object in the module and it is layered in uvm_top.

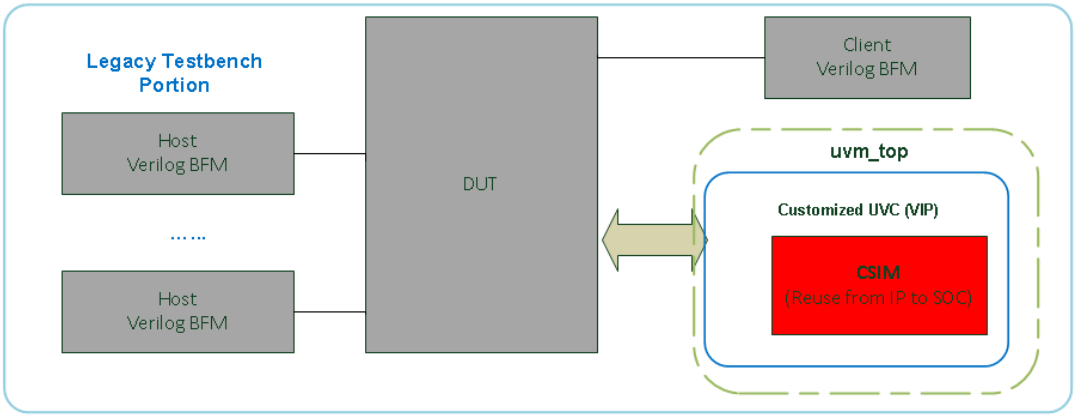


**Fig. 9. The approach of customized UVC integration in legacy TB.**

```
module tb;
   `include "customized_uvc_env_cfg.sv"
   `include "customized_uvc_env.sv"
   customized_uvc_env          uvm_vip_env;
   customized_uvc_env_cfg     uvm_vip_env_cfg
   initial begin
      uvm_vip_env_cfg = customized_uvc_env_cfg::type_id::create("uvm_vip_env_cfg", uvm_top);
      uvm_vip_env = customized_uvc_env::type_id::create("uvm_vip_env", uvm_top);
      uvm_config_db#( customized_uvc_env_cfg)::set(this," uvm_vip_env*", "cfg", uvm_vip_env_cfg);
      `uvm_info("tb", " uvm_vip_env  is created", UVM_LOW);
   end
```

**Fig. 10. The UVC integration code example.**

❖ Integrate the customized UVC in the new UVM test bench
It is as easily integrated as other UVM components shown in Fig. 11.

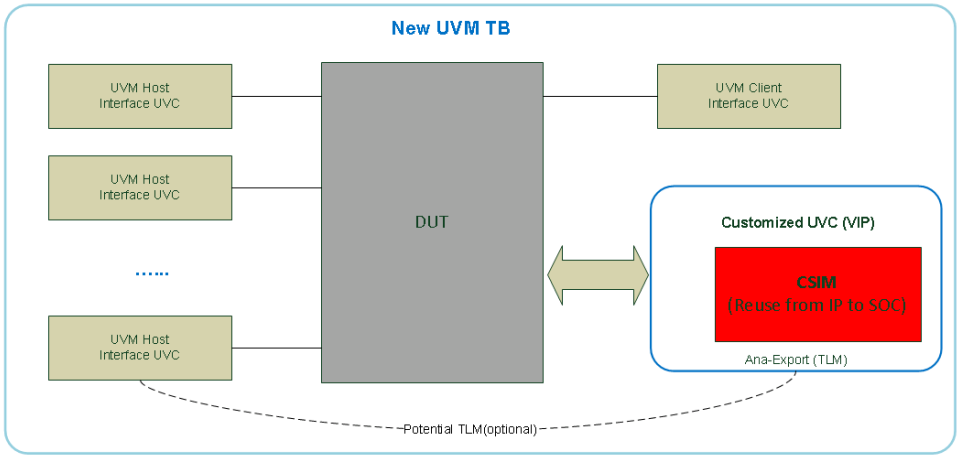


**Fig. 11. The approach of customized UVC integration in new UVM TB.**

## IV. RESULTS

In this paper, we describe a solution to develop the customized UVM VIP and embed the cycle-based C++ reference model. Comparing with legacy solution, this solution can reuse the existing CSIM model and be easily integrated into both the legacy and the new UVM testbench. The most important benefits are to boost the runtime simulation performance because of the transaction level through TLM connection and provide more friend transaction level debugging ability. It is proven to be practical, high performance, easy to integrate and adopt, and good for debugging.

## V. FUTURE WORKS

In some more special cases, the CSIM could also require other information:

- The register's value by connect to extern register related TLM.
- The result of a simple algorithm written by System Verilog (SV), it needs information from other input agents.

## ACKNOWLEDGMENT

I would like first to thank my wife (Liangliang Li) for her continued support and my daughter (Tatiana Wang) to reserve much time to me for thinking and writing.



## REFERENCES

[1] IEEE 1800-2009 SystemVerilog

[2] http://www.accellera.org/apps/org/workgroup/uvm/

[3] Roman Wang, "Wrapping Verilog Bus Functional Model (BFM) and RTL as Drivers in Customized UVM VIP Using Abstract Classes", DVCon USA 2015.