# Deploying Parameterized Interface with UVM

Wayne Yun
AMD, Inc.
1-289-695-1968
wayne.yun@amd.com

Shihua Zhang
AMD, Inc.
1-289-695-1969
shihua.zhang@amd.com

***Abstract*-SystemVerilog defines syntax of interface to ease design as well as verification. Open Verification Methodology (OVM) and Universal Verification Methodology (UVM) utilize it to connect dynamic classes and static signals. As IP and SOC designs become more and more configurable, parameterized interfaces are naturally deployed. Because parameters are static and classes are dynamic, such interfaces are challenging on the UVM side, especially when the number of parameters is large. This paper analyzes the issue, gives a few workarounds for simple cases, and details a generic solution of using class to abstract an interface. Some variants are also explored, and tested examples are listed.**

***Keywords-functional verification, DV, OVM, UVM, SOC, IP***

## I. INTRODUCTION

Re-usability, scalability, and flexibility are some important features introduced by the OVM and followed by the UVM (this paper uses UVM to refer to both methodologies). As such, the UVM is often chosen to verify configurable designs. Naturally, interfaces of these designs are parameterized, as demonstrated by the existence of parameters in Verilog modules and patterns like sig_0, sig_1, etc., in port names. Following UVM, SystemVerilog interfaces would be defined for interfaces of designs; consequently, it would be ideal to parameterize these SystemVerilog interfaces.

UVM utilizes virtual interfaces at dynamic driver and monitor classes to access to static interfaces. By SystemVerilog definition, the parameters of virtual interfaces have to be the same as instantiated parameters for assignment compatibility. These parameters are compile-time constant, which means their values cannot come from run-time configuration objects. This forces a literal number or another parameter to be used at the definition of a virtual interface. Obviously, a literal number is not in line with re-usable, scalable, and flexible verification and should not be used. The other choice, a parameter, essentially propagates parameters to higher-level testbench components until they reach the top [4]. This would cause all sorts of issues with maintenance, scalability, flexibility, etc.

This paper addresses the significant obstacles to using parameterized interfaces in UVM testbenches, especially in highly configurable designs.

## II. WORKAROUNDS

For some simple cases, the simplicity can be leveraged to make workarounds.

### A. *Parameterizing All Classes*

[4] described a working solution of propagating all parameters from interface to UVM classes, including Universal Verification Components (UVCs) and testbench.

In this solution, almost every UVM class has parameters; more maintenance work results when a new parameter is added or an existing one is modified. This practice essentially requires higher-level testbench components to have detailed knowledge of lower-level components, which is not quite compatible with the idea of encapsulation from an object-oriented programming (OOP) perspective.

### B. *Defining Super-set Signals*

This workaround defines a super, non-parameterized interface that has every signal of every potential configuration. Those unused signals are tied off at instantiation.

Generally, this workaround is less clean and creates overhead in some cases, especially when the difference in number of signals is huge among potential configurations.

### C. *Sub-interface*

Some interfaces are composed of a number of groups of signals. A group-level interface can be defined. Each group should have one virtual interface passed to driver and monitor.

This workaround will not give as neat a solution when groups are not identical to each other.

These workarounds could work for simple cases with certain degrees of compromise in cleanness. The real issue is that they are not scalable, and so a fundamentally different solution is needed for more complex cases.

## III. SOLUTION

The general solution resides in using classes to abstract parameterized interfaces, which is allowed by SystemVerilog's syntax such that class objects representing interfaces are used at the driver and monitor and the compile-time parameter is not needed. No parameter of interface is mandatory for dynamic verification components, and they can be coded independently for re-usability.

[6] introduced how to use abstract class to represent interface in SystemVerilog. In general, following OOP, a derived class can be used as its base class. Prototypes of virtual methods are defined by base class and implemented by derived classes. Such each derived class can have a different implementation but all have the same operational methods defined by base class. So the base class is an abstraction of

This paper is sponsored by AMD.

derived classes. The derived class could be parameterized -- which is allowed by SystemVerilog -- and associated with a parameterized interface. On the other hand, the base class becomes an abstract class for the interface.

[5] described an example of using a class to abstract an interface. It demonstrated only relationships between classes; connections among ports of an interface and fields of classes are missing (i.e., there is no bridge of static signals and dynamic variables), so it cannot be deployed. Solving this problem is the main contribution of this paper. Different implementations and variants are also presented.

### A. *General Solution*

Four major steps are involved in abstracting an interface: packet definition, abstract access class, concrete access class (which is derived from the abstract one), and association of interface and concrete access class. Defined packets and abstract access class are used by the driver and monitor to operate on an interface.

1. Packets (UVM sequence items) representing traffic at an interface should be defined at a level of abstraction such that no parameter is needed in its class definition. For example, the width of an address signal is defined by a parameter at the interface. In a sequence item, it can be defined to have the maximum possible width. Such higher-level checking and stimulus are isolated from implementation details. If the width is required, a field for address width can be introduced in a sequence item; this field is dynamic rather than compile-time constant.

```
// transaction
class transaction extends uvm_sequence_item;
   // abstract level definitions
   rand bit [2:0]   port_num;
   rand bit [63:0] address;
   rand bit [255:0] data;

   `uvm_object_utils_begin(transaction)
      `uvm_field_int(port_num, UVM_ALL_ON)
      `uvm_field_int(address, UVM_ALL_ON)
      `uvm_field_int(data, UVM_ALL_ON)
   `uvm_object_utils_end

   function new(string name = "transaction");
      super.new(name);
   endfunction: new
endclass: transaction
```

Example 1: Packet Definition

2. Abstraction of access to interface is done by defining a virtual access class. Virtual methods should be defined for the operations that the driver and monitor of this interface would perform. Depending on trade-offs made for overall structure, these methods could vary from setting a specific signal to sending a complete packet or transaction.

```
// intf_ctlr_base
virtual class intf_ctlr_base extends uvm_object;

   function new(string name = "intf_ctlr_base");
      super.new(name);
   endfunction: new
   // Virtual methods
   pure virtual function void set_v_intf(string interface_path);
   pure virtual task drive_transaction(transaction trans);
   pure virtual task monitor_transaction(uvm_analysis_port
#(transaction) ap);

endclass : intf_ctlr_base
```

Example 2: Abstract Class Definition

3. The concrete access class is extended from the abstract one and implements all access methods. It is parameter-aware. It is simpler to make this class have exactly the same parameters as the interface.

```
// interface
interface intf #(parameter int num_ports = 1, parameter int
address_width = 32, parameter int data_width = 128) (
   input                          clk,
   inout [num_ports-1:0]          port_valid,
   inout [address_width-1:0]    address,
   inout [data_width-1:0]         data
);
```

Example 3: Interface Definition

```
// intf_ctlr
class intf_ctlr #(num_ports = 1, address_width = 32, data_width = 128)
extends intf_ctlr_base;
   // Virtual interface to be driven
   protected virtual interface intf #(num_ports, address_width,
data_width) v_intf;

   function new(string name = "intf_ctlr");
      string intf_ctlr_path;
      super.new(name);
      intf_ctlr_path = {name, ".ctlr"};
      set_config_object("*", intf_ctlr_path, this, 0); // Set intf ctlr
   endfunction: new

   // Set virtual interface
   function void set_v_intf(string interface_path);
      uvm_resource_db #(virtual intf#(num_ports, address_width,
data_width))::read_by_name("interfaces", interface_path, v_intf);
   endfunction

   task drive_transaction(transaction trans);
      @(posedge v_intf.clk);
      v_intf.port_valid[trans.port_num] <= 1'b1;
      v_intf.address <= trans.address;
      v_intf.data <= trans.data;
      @(posedge v_intf.clk);
      v_intf.port_valid <= 'bz;
      v_intf.address <= 'bz;
```

```
      v_intf.data <= 'bz;
   endtask
   task monitor_transaction(uvm_analysis_port #(transaction) ap);
      int port_num;
      transaction trans;
      forever begin
         @(posedge v_intf.clk);
         if (v_intf.port_valid != '0) begin
            trans = transaction::type_id::create("trans");
            port_num = 0;
            while (v_intf.port_valid[port_num] !== 1)
               port_num++;
            trans.port_num = port_num;
            trans.address = v_intf.address;
            trans.data = v_intf.data;
            `uvm_info("MON_TRANS", $psprintf("Monitor received a
trans:\n%s", trans.sprint()), UVM_LOW)
            ap.write(trans);
         end
      end
   endtask

endclass : intf_ctlr
```

Example 4: Concrete Class

4. There are many valid solutions for associating the concrete access class to interface. A few cornerstones are:

   a. Concrete class can be stand-alone, instantiated with all parameters, and a virtual interface is used. Examples 1 to 11 give a complete runnable testbench.

      Driver gets a reference of base class type and uses it to send out transactions. Driver is not parameterized.

```
// driver
class driver extends uvm_driver #(transaction);
   string interface_path;

   intf_ctlr_base m_if_ctlr;

   `uvm_component_utils_begin(driver)
      `uvm_field_string(interface_path, UVM_ALL_ON)
   `uvm_component_utils_end

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

   virtual function void build_phase(uvm_phase phase);
      string intf_ctlr_path;
      uvm_object object;
      super.build_phase(phase);

      // Get interface controller
      if (get_config_object(intf_ctlr_path, object, 0))
         $cast(m_if_ctlr, object);
      else begin
         `uvm_fatal("DRV_BUILD", "Cannot find interface controller")
      end
   endfunction

   virtual task run_phase(uvm_phase phase);
      forever begin
         seq_item_port.get_next_item(req);
         `uvm_info("DRV_RUN", req.sprint(), UVM_MEDIUM);
         // drive req by interface controller
         m_if_ctlr.drive_transaction(req);
         seq_item_port.item_done(req);
      end
   endtask
endclass : driver
```

Example 5: Driver Class

Monitor gets a reference of base class type and uses its monitoring method. Monitor is not parameterized.

```
// monitor
class monitor extends uvm_monitor ;
   string interface_path;

   intf_ctlr_base m_if_ctlr;

   uvm_analysis_port #(transaction) request_ap;

   `uvm_component_utils_begin(monitor)
      `uvm_field_string(interface_path, UVM_ALL_ON)
   `uvm_component_utils_end

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

   virtual function void build_phase(uvm_phase phase);
      string intf_ctlr_path;
      uvm_object object;
      super.build_phase(phase);

      // Get interface controller
      if (get_config_object(intf_ctlr_path, object, 0))
         $cast(m_if_ctlr, object);
      else begin
         `uvm_fatal("MON_BUILD", "Cannot find interface controller")
      end

      // Pass interface_path to controller so the controller can
      // get the interface to be driven
      m_if_ctlr.set_v_intf(interface_path);
   endfunction

   virtual task run_phase(uvm_phase phase);
```

```
    m_if_ctlr.monitor_transaction(request_ap);
  endtask

endclass : monitor
```

Example 6: Monitor Class

Sequencer and agent are typical, having no parameter propagated from interface.

```
// sequencer
typedef uvm_sequencer #(transaction) sequencer;

// agent
class agent extends uvm_agent;
  sequencer m_seqr;
  driver m_drv;
  monitor m_mon;

  `uvm_component_utils(agent)

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    m_seqr = sequencer::type_id::create("m_seqr", this);
    m_drv = driver::type_id::create("m_drv", this);
    m_mon = monitor::type_id::create("m_mon", this);

  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    m_drv.seq_item_port.connect(m_seqr.seq_item_export);

  endfunction

endclass : agent
```

Example 7: Driver Class

UVC is a simple uvm_env without parameter.

```
// env
class env extends uvm_env;

  agent m_agent;

  `uvm_component_utils(env)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("ENV_NEW", "inside new()", UVM_LOW)
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_agent = agent::type_id::create("m_agent", this);
  endfunction

endclass : env
```

Example 8: UVC Class

Sequence is same as non-parameterized interface cases.

```
// packet
class packet extends uvm_sequence #(transaction);
  transaction trans;

  `uvm_object_utils_begin(packet)
    `uvm_field_object(trans, UVM_ALL_ON)
  `uvm_object_utils_end
  `uvm_declare_p_sequencer(sequencer)
  function new(string name="packet");
    super.new(name);
    trans = transaction::type_id::create("trans");
  endfunction : new

  virtual task body();
    start_item(trans);
    trans.randomize();
    finish_item(trans);
    get_response(rsp);
  endtask
endclass
```

Example 9: Sequence Class

UVM test creates a back-to-back interface UVC testbench and passes in proper string for retrieving access class.

```
// test
program automatic test;
  import uvm_pkg::*;
  `include "uvm_macros.svh"
  class test extends uvm_test;
    `uvm_component_utils(test)
    env m_env;
    env s_env;
    function new(string name, uvm_component parent);
      super.new(name, parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      // Set interface path to driver and monitor
      uvm_config_db#(string)::set(this, "m_env*.*", "interface_path",
"top.m_inst");
      m_env = env::type_id::create("m_env", this);
      uvm_config_db#(string)::set(this, "s_env*.*", "interface_path",
"top.s_inst");
      s_env = env::type_id::create("s_env", this);
    endfunction
```

```
    virtual task run_phase(uvm_phase phase);
      packet p;
      phase.raise_objection(this);
      repeat (10) begin
        p = packet::type_id::create("p");
        `uvm_info("TEST_RUN", "Start packet on m_env", UVM_LOW)
        p.start(m_env.m_agent.m_seqr);
        #20;
        p = packet::type_id::create("p");
        `uvm_info("TEST_RUN", "Start packet on s_env", UVM_LOW)
        p.start(s_env.m_agent.m_seqr);
        #20;
      end
      phase.drop_objection(this);
    endtask
  endclass: test

  initial
    run_test();
endprogram
```

Example 10: Driver Class

Interfaces are instantiated on the static side of the testbench and references to instances of access class are saved to resource database.

```
`define NUM_PORTS    8
`define ADDR_WIDTH   64
`define DATA_WIDTH   256
// top
module top;
  import uvm_pkg::*;

  reg                      clk;
  wire [`NUM_PORTS-1 : 0]    port_valid;
  wire [`ADDR_WIDTH-1 : 0]   address;
  wire [`DATA_WIDTH-1 : 0]   data;

  intf #(`NUM_PORTS, `ADDR_WIDTH, `DATA_WIDTH) m_inst(clk,
port_valid, address, data);
  intf #(`NUM_PORTS, `ADDR_WIDTH, `DATA_WIDTH) s_inst(clk,
port_valid, address, data);

  intf_ctlr #(`NUM_PORTS, `ADDR_WIDTH, `DATA_WIDTH) m_intf_ctlr =
new($psprintf("top.m_inst"));
  intf_ctlr #(`NUM_PORTS, `ADDR_WIDTH, `DATA_WIDTH) s_intf_ctlr =
new($psprintf("top.s_inst"));

  initial begin
    uvm_resource_db#(virtual intf#(`NUM_PORTS, `ADDR_WIDTH,
`DATA_WIDTH))::set("interfaces", "top.m_inst", m_inst);
    uvm_resource_db#(virtual intf#(`NUM_PORTS, `ADDR_WIDTH,
`DATA_WIDTH))::set("interfaces", "top.s_inst", s_inst);
  end
```

```
  initial begin
    clk = 1'b0;
    forever begin
      #5;
      clk = ~clk;
    end
  end
endmodule : top
```

Example 11: Testbench for Stand-alone Concrete Class

b. SystemVerilog allows a class to be defined and instantiated inside an interface. Concrete access class objects can be created inside an interface, and all parameters and a virtual interface can be used.

```
// interface
interface intf #(parameter int num_ports = 1, parameter int
address_width = 32, parameter int data_width = 128) (
  input                          clk,
  inout [num_ports-1:0]          port_valid,
  inout [address_width-1:0]      address,
  inout [data_width-1:0]         data
);
  // Definition of concrete intf ctlr, no parameters needed
  class intf_ctlr extends intf_ctlr_base;
    protected virtual interface intf #(num_ports, address_width,
data_width) v_intf;

    function new(string name = "intf_ctlr");
      string intf_ctlr_path;
      super.new(name);
      intf_ctlr_path = {name, ".ctlr"};
      set_config_object("*", intf_ctlr_path, this, 0);
    endfunction: new

    function void set_v_intf(string interface_path);
      uvm_resource_db #(virtual intf#(num_ports, address_width,
data_width))::read_by_name("interfaces", interface_path, v_intf);
    endfunction

    task drive_transaction(transaction trans);
      @(posedge v_intf.clk);
      v_intf.port_valid[trans.port_num] <= 1'b1;
      v_intf.address <= trans.address;
      v_intf.data <= trans.data;
      @(posedge v_intf.clk);
      v_intf.port_valid <= 'bz;
      v_intf.address <= 'bz;
      v_intf.data <= 'bz;
    endtask

    task monitor_transaction(uvm_analysis_port #(transaction) ap);
      int port_num;
      transaction trans;

      forever begin
```

```
            @(posedge v_intf.clk);
            if (v_intf.port_valid != '0) begin
               trans = transaction::type_id::create("trans");
               port_num = 0;
               while (v_intf.port_valid[port_num] !== 1)
                  port_num++;
               trans.port_num = port_num;
               trans.address = v_intf.address;
               trans.data = v_intf.data;
               `uvm_info("MON_TRANS", $psprintf("Monitor received a
trans:\n%s", trans.sprint()), UVM_LOW)
               ap.write(trans);
            end
         end
      endtask
   endclass : intf_ctlr

   // Instantiate interface controller
   intf_ctlr m_intf_ctlr = new($psprintf("%m"));

endinterface : intf
```

Example 12: Concrete Class in Interface

c. Threads started by an interface can access its ports. Then transactions can be made and passed though the concrete access class.

```
// interface
interface intf #(parameter int num_ports = 1, parameter int
address_width = 32, parameter int data_width = 128) (
   input                          clk,
   inout [num_ports-1:0]          port_valid,
   inout [address_width-1:0]      address,
   inout [data_width-1:0]         data
);
   // intf_ctlr
   class intf_ctlr extends intf_ctlr_base;

      protected virtual interface intf #(num_ports, address_width,
data_width) v_intf;

      function new(string name = "intf_ctlr");
         string intf_ctlr_path;
         super.new(name);
         intf_ctlr_path = {name, ".ctlr"};
         set_config_object("*", intf_ctlr_path, this, 0);
      endfunction: new

      function void set_v_intf(string interface_path);
         uvm_resource_db #(virtual intf#(num_ports, address_width,
data_width))::read_by_name("interfaces", interface_path, v_intf);
         -> v_intf_set_done;
      endfunction

      function void set_analysis_port(uvm_analysis_port #(transaction)
ap);
      endfunction

      task drive_transaction(transaction trans);
         @(posedge v_intf.clk);
         v_intf.port_valid[trans.port_num] <= 1'b1;
         v_intf.address <= trans.address;

         v_intf.data <= trans.data;
         @(posedge v_intf.clk);
         v_intf.port_valid <= 'bz;
         v_intf.address <= 'bz;
         v_intf.data <= 'bz;
      endtask

      task monitor_transaction();
         int port_num;
         transaction trans;

         fork
            wait(ap_set_done.triggered);
            wait(v_intf_set_done.triggered);
         join

         forever begin
            @(posedge v_intf.clk);
            if (v_intf.port_valid != '0) begin
               trans = transaction::type_id::create("trans");
               port_num = 0;
               while (v_intf.port_valid[port_num] !== 1)
                  port_num++;
               trans.port_num = port_num;
               trans.address = v_intf.address;
               trans.data = v_intf.data;
               `uvm_info("MON_TRANS", $psprintf("Monitor received a
trans:\n%s", trans.sprint()), UVM_LOW)
               ap.write(trans);
            end
         end
      endtask

   endclass : intf_ctlr

   intf_ctlr m_intf_ctlr = new($psprintf("%m"));

   // Start monitor thread in interface
   initial begin
      m_intf_ctlr.monitor_transaction();
   end

endinterface : intf
```

Example 13: Thread Started by Interface

### *B. Variants*

Drivers and monitors can also be implemented in different manners.

1. Driver or Monitor Function Level

Methods of abstract class can be defined at the packet layer like send or receive functions. All preceding examples are at this level.

2. Signal Access Level

Methods of abstract class are defined as operations on signals. Typically, they are set and get functions.

```
// intf_ctlr_base
virtual class intf_ctlr_base extends uvm_object;

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

   // Provide signal level virtual methods
   pure virtual function void set_v_intf(string interface_path);
   pure virtual task wait_clk_posedge();
   pure virtual function void get_port_valid(output logic[7:0] port_valid);
   pure virtual function void get_address(output logic[63:0] address);
   pure virtual function void get_data(output logic[255:0] data);
   pure virtual function void set_port_valid(input logic[2:0] port_num);
   pure virtual function void set_address(input logic[63:0] address);
   pure virtual function void set_data(input logic[255:0] data);
   pure virtual function void release_port_valid();
   pure virtual function void release_address();
   pure virtual function void release_data();
endclass : intf_ctlr_base

// interface
interface intf #(parameter int num_ports = 1, parameter int
address_width = 32, parameter int data_width = 128) (
   input                          clk,
   inout [num_ports-1:0]          port_valid,
   inout [address_width-1:0]      address,
   inout [data_width-1:0]          data
);
   // intf_ctlr
   class intf_ctlr extends intf_ctlr_base;

      protected virtual interface intf #(num_ports, address_width,
data_width) v_intf;

      function new(string name = "intf_ctlr");
         string intf_ctlr_path;
         super.new(name);
         intf_ctlr_path = {name, ".ctlr"};
         set_config_object("*", intf_ctlr_path, this, 0);
      endfunction: new

      function void set_v_intf(string interface_path);
         uvm_resource_db #(virtual intf#(num_ports, address_width,
data_width))::read_by_name("interfaces", interface_path, v_intf);
      endfunction

      virtual task wait_clk_posedge();
         @(posedge v_intf.clk);
      endtask

      virtual function void get_port_valid(output logic[7:0] port_valid);
         port_valid = v_intf.port_valid;
      endfunction
      virtual function void get_address(output logic[63:0] address);
         address = v_intf.address;
      endfunction
      virtual function void get_data(output logic[255:0] data);
         data = v_intf.data;
      endfunction
      virtual function void set_port_valid(input logic[2:0] port_num);
         v_intf.port_valid[port_num] = 1;
      endfunction
      virtual function void set_address(input logic[63:0] address);
         v_intf.address = address;
      endfunction
      virtual function void set_data(input logic[255:0] data);
         v_intf.data = data;
      endfunction
      virtual function void release_port_valid();
         v_intf.port_valid = 'bz;
      endfunction
      virtual function void release_address();
         v_intf.address = 'bz;
      endfunction
      virtual function void release_data();
         v_intf.data = 'bz;
      endfunction

   endclass : intf_ctlr
   intf_ctlr m_intf_ctlr = new($psprintf("%m"));

endinterface : intf
```

Example 14: Signal-level Access

3. Drivers or Monitors Themselves

The UVM driver and monitor could be parameterized in the same way as the interface and connected to the rest of the dynamic testbench through an abstract class.

```
// interface
interface intf #(parameter int num_ports = 1, parameter int
address_width = 32, parameter int data_width = 128) (
   input                          clk,
   inout [num_ports-1:0]          port_valid,
   inout [address_width-1:0]      address,
   inout [data_width-1:0]         data
);
   // intf_ctlr
   class intf_ctlr extends intf_ctlr_base;

      function new(string name = "intf_ctlr");
         string intf_ctlr_path;
```

```
        super.new(name);
        intf_ctlr_path = {name, ".ctlr"};
        set_config_object("*", intf_ctlr_path, this, 0);
      endfunction: new

      function driver_base create_driver(uvm_component comp);
        driver_base m_drv_l;
        driver#(num_ports, address_width, data_width) m_drv =
driver#(num_ports, address_width, data_width)::type_id::create("m_drv",
comp);
        m_drv_l = m_drv;
        return m_drv_l;
      endfunction

      function uvm_monitor create_monitor(uvm_component comp);
        monitor#(num_ports, address_width, data_width) m_mon =
monitor#(num_ports, address_width,
data_width)::type_id::create("m_mon", comp);
        `uvm_info("INTF_CTLR", $psprintf("num_ports = %0d,
address_width = %0d, data_width = %0d\n", num_ports, address_width,
data_width), UVM_LOW)
        return m_mon;
      endfunction

   endclass : intf_ctlr
   intf_ctlr m_intf_ctlr = new($psprintf("%m"));

endinterface : intf
```

Example 15: Driver and Monitor in Interface

There are variants of these alternatives; we list only some of most important combinations, and there are many valid combinations and derivatives. Trade-offs should be made for specific implementation.

### C. *Structure to Avoid*

One code structure was found not to work, at least with the simulators used in this work: to access interface signals from class defined inside the interface. Though no compile error was given, signals were not driven properly when the interface was instantiated multiple times.

## IV. RESULTS

OVM/UVM testbenches were developed utilizing the methods described. The testbenches were run with OVM 2.1.2 and UVM 1.1a. The simulator was Synopsys VCS 2010.06 and 2011.12.

Tens of parameterized interfaces were deployed. Those parameters existed only at interfaces. No parameter was used in any sequence item or OVC/UVC. Such higher-level checking and stimulus are re-used easily when the DUT is re-parameterized, and even when it is re-structured. Maintenance is also reduced.

Using class to abstract an interface overcomes the lack of interface abstraction in SystemVerilog. The full re-usability, scalability, and flexibility potential of UVM can be reached utilizing the methods described in this paper.

## REFERENCES


[1] IEEE 1800-2009 IEEE Standard for SystemVerilog, New York, IEEE, 2009.

[2] Accellera Systems Initiative, *Home – Accellera Systems Initiative*, www.accellera.org.

[3] Accellera Organization, Inc., *UVM World: Universal Verification Methodology*, www.uvmworld.org.

[4] Bryan Ramirez and Michael Horn, "Parameters and OVM – Can't They Just Get Along?" DVCon 2011.

[5] Shashi Bhutada, "Polymorphic Interfaces," Verification Horizons, Volume 7, Issue 3, 2011.

[6] David Rich and Jonathan Bromley, "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches" DVCon 2008.