A Generic UVM Agent with Fine-grain Command-line Configuration

Anirban Bhattarcharjee, Sumanth Gudlavalleti, Nikhil Kikkeri,
Hui Shi, Sandra Shih, Daniel Wei

Oracle, Inc.

Santa Clara, CA, USA

http://www.oracle.com

ABSTRACT

The number and diversity of interfaces in modern day SoCs present a challenge for verification environments. Verification methodologies based on object-oriented programming have mitigated the risk in developing and maintaining testbench code. However, ever-increasing RTL complexity and tight verification schedules are a motivation for pushing the envelope for testbench code reuse. This paper describes a parameterized UVM based component library (driver, monitor, configuration object, base sequence, and agent) and the related techniques to cater to interfaces of different widths, which handle different packet types, and, which can work on different interface protocols. The same set of components could be used across all the interfaces scaling up seamlessly to the number of interfaces of each type. The interface protocol would be implemented at the sequence level thereby removing the need for creating a new interface component for each interface. Finally, the paper also describes a technique to fine-tune run-time configuration via certain enhancements to the UVM command-line processor.

Table of Contents

Table of Figures

1. Introduction

Testbench environments implemented in SystemVerilog [1] are primary vehicles for finding bugs in present-day logic designs. Techniques for developing such verification environments are well documented in [2]. Testbench code reuse is an essential feature of SystemVerilog that can reduce testbench development times. The motivation to standardize features in the different base-class libraries finally led to the Universal Verification Methodology (UVM) [3].

In UVM, verification code for stimulus generation, run-time configuration, sampling and/or driving a common set of wires is organized in a class called an *agent*. The testbench environment consists of any number of such agents required to verify the design-under-test. Inheriting UVM base classes is typically the first level of code reuse. In this stage, the sub-classes are used to instantiate the various components inside an agent/environment, to implement the transaction-level protocol, to connect the various components so that generated stimulus objects can be used to exercise the wires, sample the wires and package them into transaction objects that are examined by a checking class etc.

The next level of reuse is an area of particular interest, as challenges addressed in this area help in faster deployment of testbench code. Reusable VIPs based on parameterized interfaces described in [4] and the concept of abstract BFMs described in [5] are a couple of examples. Both examples deal with easing the burden introduced by different interfaces. The former assumes similar protocol for all interfaces but allows for different signal widths on different interfaces. The latter describes an abstract API that must be implemented in order to achieve the desired driving/sampling behavior. Signal-level complexity is hidden away in the concrete BFM thereby achieving reuse of components driving and/or sampling signals.

In this paper, we attempt to provide our take on UVM component reuse under different situations like interface and protocol differences. These differences either manifest themselves as signals of different widths as described in the previous paragraph, or, in the manner the signals are driven or sampled. An example of the latter is a protocol that requires multiple cycles to drive address and data. The paper describes a packet-level API that must be implemented for different interfaces. In combination with this API, protocol-level complexity is abstracted out from the driver and monitor components and implemented in the UVM sequence class that generates the random transactions. This abstraction yields a generic driver, monitor and sequencer that can potentially be used on any interface.

Finally the paper discusses a technique to tune the run-time configuration at the agent-level granularity in a situation where a testbench may have multiple agent instances. The tuning may be required either to constrain the randomization in different ways on different agents, or to craft a directed test.

2. Handling Parameterized Interfaces and Signal Differences

The Abstract BFM class described in [5] is used as a starting point. This class provides a set of pure virtual methods to drive and monitor the interface. A sub-class of the abstract BFM class gets implemented in the actual SystemVerilog interface.

Subsequently, all interface signals are classified into two groups: control and data. The set of signals which provides a validity condition is connected to the control part of the interface (denoted as ctrl) and the remaining signals are connected to the data (denoted as data) part of the interface definition.

The interface is parameterized by its signal widths, and the interface methods use vector variables which are wide enough (the widths are specified as constants) to work for all interfaces in a DUT. Note that in typical usage, the interface can be parameterized to a size much smaller than the constant vector size. So it would appear as though one is assigning a value in a wide vector to a variable in a narrow bus. VCS seems fine with this sort of assignment. There can be a top level or an agent level build-time option (denoted as `V_MAX_DATA_SIZE , `V_MAX_CTRL_SIZE) that gets updated whenever the agent gets used on a wider interface. The driver and the monitor make use of the methods in the interface. The fixed width vector works seamlessly with the interface whose signal widths are parameterized irrespective of the width differences.

```
virtual class AbstractIntfClass extends uvm_object;

  pure virtual task drive(logic [`V_MAX_DATA_SIZE-1:0] data,
                          logic [`V_MAX_CTRL_SIZE-1:0] ctrl);

  pure virtual task monitor(ref logic [`V_MAX_DATA_SIZE-1:0] data,
                            ref logic [`V_MAX_CTRL_SIZE-1:0] ctrl);

  pure virtual task wait_cycle(int cycles);
  pure virtual task driveIdle();
  pure virtual task driveX();
  ......

endclass: AbstractIntfClass

interface PortIntf #(int DATA_WIDTH = `V_MAX_DATA_SIZE,
                     int CTRL_WIDTH = `V_MAX_CTRL_SIZE)
                     (input bit                   clk,
                      inout wire [DATA_WIDTH-1:0]  data,
                      inout wire [CTRL_WIDTH-1:0]  ctrl);

  clocking driverCB @(posedge clk);
    output data;
    output ctrl;
  endclocking:driverCB

  clocking monitorCB @(posedge clk);
    input data;
    input ctrl;
  endclocking:monitorCB

  class ActualIntfClass extends AbstractIntfClass;
    function new(string name="");
      super.new(name);
    endfunction

    task drive(logic [`V_MAX_DATA_SIZE-1:0] data,
               logic [`V_MAX_CTRL_SIZE-1:0] ctrl);
      driverCB.ctrl <= ctrl;
      driverCB.data <= data;
      @(driverCB);
      driverCB.ctrl <= 0;
      driverCB.data <= 0;
    endtask
     task monitor(ref logic [`V_MAX_DATA_SIZE-1:0] data,
                 ref logic [`V_MAX_CTRL_SIZE-1:0] ctrl);
       wait(|(monitorCB.ctrl) == 1'b1);
       data = monitorCB.data;
       ctrl = monitorCB.ctrl;
     endtask

  endclass: ActualIntfClass

  ActualIntfClass IntfInst;

  function AbstractIntfClass getIntfInst();
    IntfInst=new("concrete_inst");
    return IntfInst;
  endfunction

endinterface: PortIntf
```

The packet-level API consists of functions required to pack the packet members into either a data or a control vector. Similarly, sampled bus vectors can be unpacked into packet members. Additionally, a monitor task in the interface is provided to sample the desired signal transitions before unpacking the vector. This coding style, for the most part, is similar to [5], but differs in consolidation of all signals into data and control.

```
virtual class PacketBase extends uvm_sequence_item;

  pure virtual function void bitsPackData(ref logic[`V_MAX_DATA_SIZE-1:0]
data);
  pure virtual function void bitsPackCtrl(ref logic[`V_MAX_CTRL_SIZE-1:0]
ctrl);
  pure virtual function void bitsUnPackData(logic [`V_MAX_DATA_SIZE-1:0]
data);
  pure virtual function void bitsUnPackCtrl(logic [`V_MAX_CTRL_SIZE-1:0]
ctrl);
  pure virtual function string psdisplay();
  .....

  int PktDelay;

endclass: PacketBase
```

The following code illustrates the application of pack and unpack methods in the driver and monitor respectively.

```
class PortDriver #(type T=PacketBase) extends uvm_driver#(T);
 `uvm_component_param_utils(PortDriver#(T))
  //Contains the virtual interface handle
  protected PortAgentConfigInfo Cfg;

  logic [`V_MAX_DATA_SIZE-1:0] data;
  logic [`V_MAX_CTRL_SIZE-1:0] ctrl;
  ......

  task run_phase(uvm_phase phase);
    T trans;
    logic [`V_MAX_DATA_SIZE-1:0] data;
    ......
    seq_item_port.get_next_item(trans);
    trans.bitsPackData(data);
    trans.bitsPackCtrl(ctrl);
    Cfg.absIntf.wait_cycle(trans.PktDelay);
    Cfg.absIntf.drive(data, ctrl);
    ......

  endtask:run_phase

endclass: PortDriver


class PortMonitor #(type T=PacketBase) extends uvm_monitor;
 `uvm_component_param_utils(PortMonitor#(T))

  protected PortAgentConfigInfo Cfg;
  uvm_analysis_port #(T) aport;
  logic [`V_MAX_DATA_SIZE-1:0] data;
  logic [`V_MAX_CTRL_SIZE-1:0] ctrl;
  ......

  task run_phase(uvm_phase phase);
     T samplePkt;

    forever begin
      Cfg.absIntf.wait_cycle(1);
      Cfg.absIntf.monitor(data, ctrl);
      samplePkt = T::type_id::create("samplePkt",this);
      samplePkt.bitsUnPackCtrl(ctrl);
      samplePkt.bitsUnPackData(data);

      //Write to Analysis port
      aport.write(samplePkt);

    end
  endtask
endclass: PortMonitor
```

The PortAgentConfigInfo serves as a placeholder for the virtual interface (contained within the BFM) and configuration options (also called “knobs”) that can be used by other classes in an agent. The knobs are read in by calling the setConfigParams function. The process of how the knobs are read will be described in section 5. The UVM way of using the uvm_config_db to set and get virtual interfaces and/or configuration objects for different agents is assumed and not shown in the above example.

```
class PortAgentConfigInfo extends uvm_object;
  AbstractIntfClass absIntf;

  function void setConfigParams(string context=””);

endclass: PortAgentConfigInfo


class PortAgent #(type T=PacketBase ) extends uvm_agent;

  PortAgentConfigInfo Cfg;

  `uvm_component_param_utils_begin(PortAgent#(T))
    `uvm_field_object(Cfg, UVM_DEFAULT)
  `uvm_component_utils_end

  PortDriver #(T) Driver;
  PortMonitor #(T) Monitor;
  uvm_sequencer #(T) Sequencer;
  ......

endclass: PortAgent
```

In the above example, driving and monitoring are shown to be a single cycle operation inside the ActualIntfClass which is a sub-class of AbstractIntfClass. It is expected that the testbench completes the implementation of the functions inside a sub-class of PacketBase. The code is now a convenient starting point which we can reuse to implement agents for any interface. We illustrate this approach with the help of two examples as shown in the next section.

3. Handling Multi-cycle Protocols

The generic agent is protocol agnostic where all the signals that need to be driven on the bus are driven in a single cycle. Cycle accuracy is preserved inside a UVM sequence based stimulus. However, the sequence code is still abstract and does not deviate significantly from the standard practice. Two examples are shown in this section: a simple multi-cycle protocol and an enhanced multi-cycle protocol.

3.1 Simple Multi-cycle Protocol

The generic agent uses a single cycle protocol. If the same protocol extends to multiple cycles on a different interface, it is possible to re-use the generic agent on that interface. The protocol differences are handled at the sequence level.

In this example protocol, there is a control/address phase and a data phase. The Master (modeled by the generic agent) drives the address and command (READ/WRITE) in the address phase along with the ‘advalid’ signal. The control phase is followed by the data phase. In case of WRITE, the Master drives the address followed by data in back to back cycles, and ‘adValid’ is asserted during these two cycles. The ‘addrDataPhase’ signal is driven by the Master to differentiate between address and data phases. In case of READ, the Slave (or the DUT) returns the data on the dataOut bus in any cycle after the Master sets the control. The ‘dvalid’ signal qualifies the dataOut bus on the response interface.

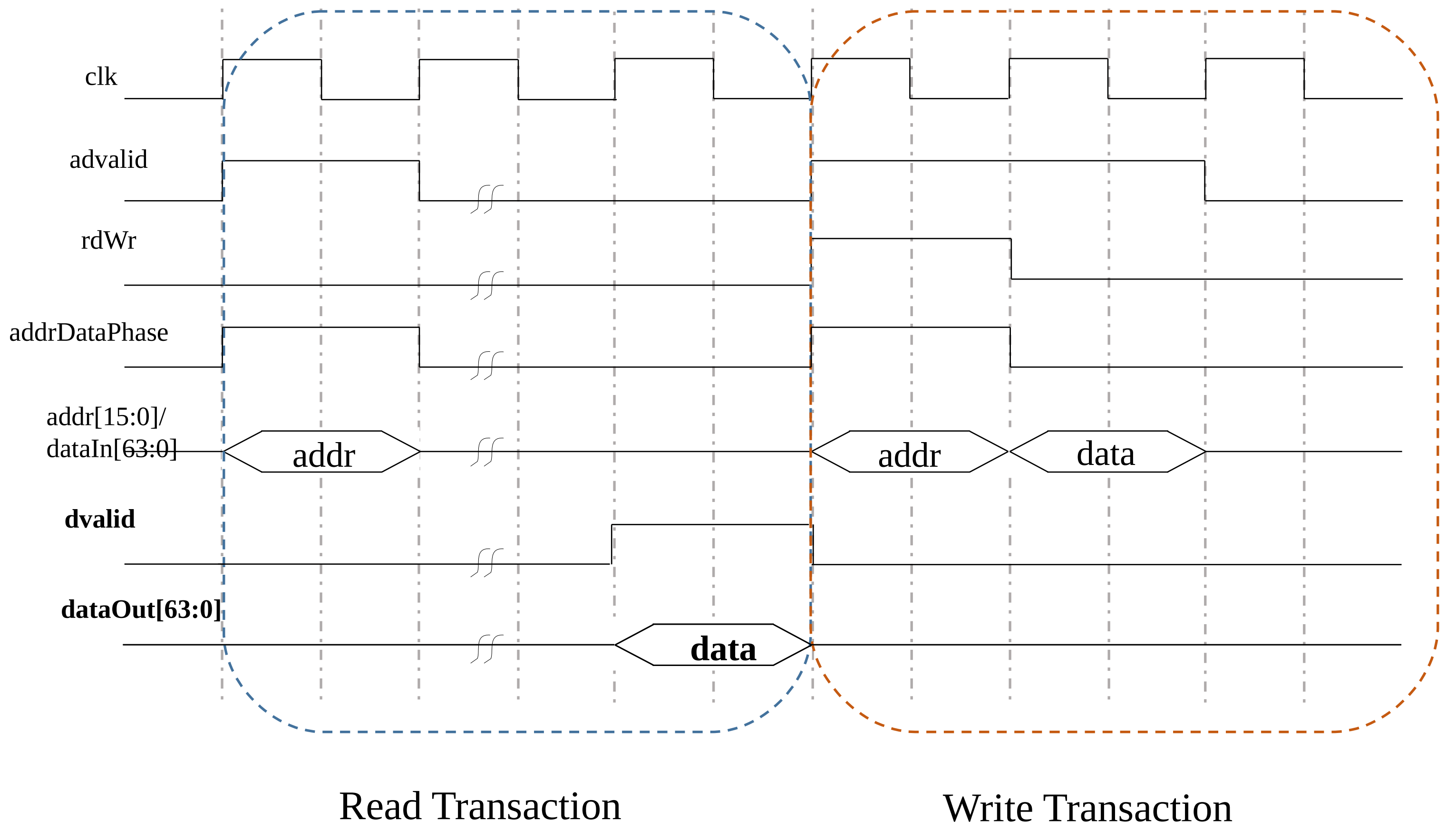

Figure 1. Single cycle READ and two-cycle WRITE.

The Request Interface (reqIntf) is used by the active agent to drive the relevant signals into the DUT. The Response Interface (respIntf) is used by agents in passive mode to sample the dataOut.

```
module tb_top
  PortIntf #(66,1) reqIntf (.clk (clk),
                            .ctrl (advalid),
                            .pkt ({rdWr,addrDataPhase,dataIn}));
  PortIntf #(64,1) respIntf (.clk ( clk),
                             .ctrl(dvalid),
                             .pkt({dataOut}))

  //set interfaces in uvm_config_db
  initial begin
    uvm_config_db #(virtual PortIntf #(66,1)):: set(uvm_root::get,"*",
                                                     "input_if",reqIntf);


    uvm_config_db #(virtual PortIntf #(64,1)::set(uvm_root::get,"*",
                                                     "output_if",respIntf);
   end
endmodule
```

The test component is responsible for generating a random test configuration and creating the test environment. The configuration elements reside in a configuration class and follows the same hierarchy as the environment. That is, there is a top-level configuration class that is responsible for creating the agent-level configuration instances, which in turn create configuration instances for agents at the next lower level. For convenience, it is a sub-class of PortAgentConfigInfo.

```
class ConfigInfo extends PortAgentConfigInfo;
  `uvm_object_utils(ConfigInfo)

  PortAgentConfigInfo reqAgt, respAgt;

  function new(string name=”ConfigInfo”);
    super.new(name);
    reqAgt = PortAgentConfigInfo::type_id::create(“Req”);
    respAgt = PortAgentConfigInfo::type_id::create(“Resp”);
  endfunction: new

  .........

  endclass: ConfigInfo
```

The test creates the configuration object before it creates the environment. The environment has a handle to the configuration class through which it can access the agent-level configuration instances. When the agents get instantiated in the environment, the corresponding packet types and interface handles are set as shown below.

```
class Env extends uvm_env;
  .......
  PortAgent#(ReqPacket) reqAgt;
  PortAgent#(RespPacket) respAgt;
  PortIntf #(66,1) input_if;
  PortIntf #(64,1) output_if;
  ConfigInfo Cfg;
  .......
  .......

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

     uvm_config_db #(PortIntf #(66,1))::get(this,””,”input_if”,input_if);
     uvm_config_db #(PortIntf #(64,1))::get(this,””,”output_if”,output_if);

     Cfg.reqAgt.absIntf = input_if.getIntfInst();
     Cfg.respAgt.absIntf = output_if.getIntfInst();

     uvm_config_db #(PortAgentConfigInfo)::set(this,”reqAgt”,
                                          ”Cfg”,Cfg.reqAgt);
     uvm_config_db #(PortAgentConfigInfo)::set(this,”respAgt”,
                                          “Cfg”,Cfg.respAgt);

    //Set reqAgt as UVM_ACTIVE, respAgt as UVM_PASSIVE
  endfunction
  .......
endclass
```

The packet classes need to implement the pack and unpack methods as per the protocol specifications.

```
class ReqPacket extends PacketBase;
  `uvm_object_utils(ReqPacket)
  bit valid;
  bit rdWr;
  bit addrDataPhase; // Address or Data Phase
  bit[15:0] addr;
  bit[63:0] data;
  bit cmd;
  .....

  //Implement the Pack/Unpack functions as per the protocol
  function void bitsPackData(ref logic[`V_MAX_DATA_SIZE-1:0] data);
    if(addrDataPhase)
      data = {rdWr,addrDataPhase, 48'h0,addr};
    else
      data = {rdWr,addrDataPhase, data};
  endfunction

  function void bitsPackCtrl(ref logic[`V_MAX_CTRL_SIZE-1:0] ctrl);
    ctrl = valid;
  endfunction

  function void bitsUnPackData(logic [`V_MAX_DATA_SIZE-1:0] data);
    rdWr = data[65];
    addrDataPhase = data[64];
    if( addrDataPhase == 1'b1)
       addr = data[15:0];
    else
       data = data[63:0];
  endfunction

  function void bitsUnPackCtrl(logic [`V_MAX_CTRL_SIZE-1:0] ctrl);
    valid = ctrl;
  endfunction

  function string psdisplay();
    ......
  endfunction
endclass


class RespPacket extends PacketBase;
  `uvm_object_utils(RespPacket)
  bit valid;
  bit[63:0] data;
  int cmd;
  .....
  //Implement  bitsPackData,  bitsPackCtrl,  bitsUnPackData,  bitsUnPackCtrl,
psdisplay methods as per the protocol

endclass
```

The Read sequence implements the control phase of the READ operation.

```
class ReadSequence extends uvm_sequence #(ReqPacket);
  `uvm_object_utils(ReadSequence)
  ReqPacket req;
  bit rdWr;

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

  task body();
    `uvm_create(req);
    start_item(req);
    req.randomize();
    req.cmd = 0; //READ command
    req.valid = 1'b1;
    finish_item(req);
  endtask
endclass: ReadSequence
```

The Write sequence implements the two phases of the WRITE operation. It sends the command in the first cycle and the data in the next cycle.

```
class WriteSequence extends uvm_sequence #(ReqPacket);
  `uvm_object_utils(WriteSequence)
  ReqPacket req;
  bit rdWr;
  function new(string name = "WriteSequence");
    super.new(name);
  endfunction : new

  task body();
  //first cycle of a two cycle write request
  //The command is sent first followed by the data
    `uvm_create(req);
    start_item(req);
    ...... //Randomize packet,
    ...... //set the relevant control fields
    ...... //of the packet for the Address Phase
    finish_item(req);


//second cycle
    `uvm_create(req);
    start_item(req);
    ...... // Randomize packet,
    req.PktDelay = 0;  //write data follows write request the very next cycle
    ...... //  set any other attributes
    finish_item(req);
  endtask

endclass: WriteSequence
```

This is an example for a simple multi-cycle protocol that a control phase is followed by a data phase. Use of the agent on another protocol variation is described in next section.

3.2 A More Complicated Multi-cycle Protocol

Figure 2 shows a hypothetical protocol used on a memory agent where the address for a read or a write could go in the same cycle as the write data for a previous write request.

The read response timing is shown for the sake of completeness to happen one cycle after the request. Variations on this protocol are fairly common in the industry.

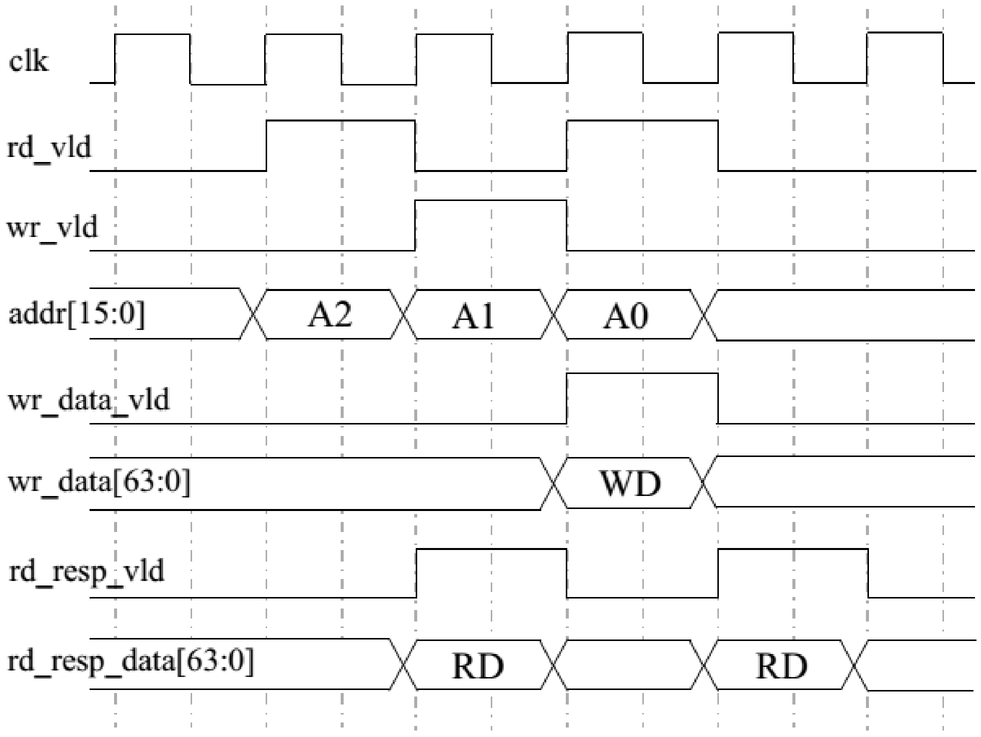

Figure 2. Read/Write requests driven concurrently with write data

As shown in Figure 2, requests could be sent concurrent to the data for a previous write transaction. A common solution to this is to employ separate threads within the driver component to drive requests/address and data. But now that the generic agent requires the protocol implementation within the sequence.

To start, as usual the packet API is implemented.

```
class MemPacket extends PacketBase;
  `uvm_object_utils(MemPacket)
  typedef enum {READ,WRITE,IDLE} type_e;
  rand bit [63:0] wr_data;
  rand bit [15:0] address;
  bit [63:0] rd_resp_data;
  bit wr_vld=1'b0;
  bit rd_vld=1'b0;
  bit wr_data_vld=1'b0;
  bit rd_resp_vld=1'b0;
  rand type_e trans_type;

  constraint c_wr_data_zero_for_reads {
    (trans_type == READ) -> (wr_data == 0);
  }

  constraint c_idle_packet {
    (trans_type == IDLE) -> ({wr_data,address} == 0); }

  ....
  //Implement  bitsPackData,  bitsPackCtrl,  bitsUnPackData,  bitsUnPackCtrl,
psdisplay methods as per the protocol

  //customized copy function for deep-copying of items
  function do_copy(uvm_sequence_item rhs);
    ......
  endfunction: do_copy

endclass: MemPacket
```

Next comes the sequence. The assumption is that we are sending 100 transactions. Since the write takes two cycles, and the assumption being that a valid request could appear on the bus at the same time as write data, a queue is maintained to hold the write until the following cycle. At the top of the loop a new transaction checks the write data queue and if it is not empty it copies over the write data of the saved transaction before being driven into the driver. In case of back-to-back writes, a deep copy is done to ensure that the contents of the current packet stay intact before being pushed into the queue.

```
class MemorySequence extends uvm_sequence;
  `uvm_object_utils(MemorySequence)

  MemPacket WriteDataQueue[$]; //queue to hold write data for a previous
transaction
  MemPacket req;
  int trans_cnt=0;

  ....

  task body();
    do begin
      `uvm_create(req)
      start_item(req);

      //if there are some transactions to send
      if( trans_cnt < 100) begin
        req.randomize();  //then randomize
        trans_cnt++; //increment number of transactions
      end
      case(req.trans_type)  //if read or write
        MemPacket::READ: begin
          //set rd_vld to 1
        end
        MemPacket::WRITE: begin
          //create and deep copy
          //the current write transaction
          //to use write data attribute in next cycle.
          //add it to the queue and set wr_vld to 1
        end
      endcase

      //Send write data for the write request sent in previous cycle.
      //The write data to be sent is in the queue of stored packets.
      //After copying, pop the top of queue

      finish_item(req);
    end while (trans_cnt < 100 || WriteDataQueue.size);

  endtask: body

endclass: MemorySequence
```

This method of wiggling the DUT pins is sufficiently general and it does not hamper the creation of more interesting scenarios to meet coverage goals.

After this the remainder of the testbench and the hookup with the design-under-test follows in the usual fashion as described in 3.1

4. Higher Level Agent

A higher level agent is usually necessary when there is a need to control multiple streams of traffic. Instead of creating one set of components for each interface, it is better to write two sets of agents. The lower level agent would be generic enough to handle all variations of a single stream of traffic. The higher level agent can be parameterized to have two integer variables to define how many instances of the lower level agent are required in Active and Passive modes. This is a common case in interconnects where many neighboring modules have similar interfaces and each interface has many streams of parallel traffic. The same higher level agent can then be instantiated for all higher level interfaces (each having multiple streams) just by passing suitable values for the two parameters.

```
class IntfAgent #(int numInPortAgts=0,
                  int numOutPortAgts=0) extends uvm_agent;
.....
PortAgent InPortAgt[numInPortAgts];
PortAgent OutPortAgt[numOutPortAgts];
.....
function void IntfAgent::build_phase(uvm_phase phase);
  super.build_phase(phase);

if(!uvm_config_db #(IntfAgentConfigInfo #(numInPortAgts,
numOutPortAgts))::get(this, "", "configInfo", cfg))
    `uvm_error("IntfAgent",{"cfg not found for: ",get_full_name()})

//Create Agent, Set the lower level CFG
for(int i = 0; i < numInPortAgts; i++) begin
 InPortAgt[i] = PortAgent::type_id::create($sformatf("InPortAgt[%0d]", i),
this);
```

```
  uvm_config_db #(PortAgentConfigInfo)::set(this,
                                           $sformatf("InPortAgt[%0d]", i),
                                                 "Cfg", cfg.InPortAgtCfg[i]);
end

for(int i = 0; i < numOutPortAgts; i++) begin
 OutPortAgt[i] = PortAgent::type_id::create($sformatf("OutPortAgt[%0d]", i),
this);
 uvm_config_db #(PortAgentConfigInfo)::set(this, $sformatf("OutPortAgt[%0d]",
i), "Cfg", cfg.OutPortAgtCfg[i]);
end
.....
endfunction
.....
endclass
```

5. Command-line Control of Multiple Stimulus Instances inside Multiple Agent Instances

A verification environment could have multiple instances of the same agent hooked up to multiple interfaces in the same design-under-test. In that case it might be desirable to tune multiple sequence instances differently from each other for various reasons. The reason could be constraining the randomization differently in multiple sequence instances or to come up with a hand-crafted "directed" test. Usually this is done in code by writing a separate test class or by specifying command-line arguments. In the latter case we felt it best to augment the UVM command-line processor with additional capabilities.

One recommendation as described in [6] is to use a configuration object to define all run-time knobs. This configuration class is of type uvm_object. Classes derived from uvm_object have no notion of hierarchy unlike those that are derived from uvm_component. However, this limitation runs counter to our intention of being able to control different sequences running on different agents. These sequences are sub-classes of uvm_sequence. Hence some contextual information has to be part of the run-time configuration that will work with sub-classes of uvm_object.

Using the capabilities of UVM command-line processor and the UVM string functions allow us to parse command-line arguments in the way we intend. The enhanced command-line processor (called CommandLineProcessor) is a singleton class that defines its own API, and which allows the testbench writer to process command-line arguments from any class regardless of whether it is a sub-class of uvm_component or uvm_object. The API provides a way to specify an optional context. The context is some information that is used to identify one class instance from another; say one instance of a driver class from another instance of the same driver class. The contextual information used is a string and does not necessarily have to relate to the UVM hierarchy. All that matters is that consistency is maintained between the command-line option and the command-line processor that is parsing it. The enhancements inside CommandLineProcessor target the most frequent use-cases.

The following code is an example of using CommandLineProcessor. Only salient aspects of this class are highlighted below. Everything else is left as an exercise for the reader.

```
static class CommandLineProcessor extends uvm_object;
   .....
```

```
//wrapper functions which use UVM commandline processor.
//read command-line and do a string match using UVM string match function
//The UVM string match function can match the wildcard “*”

function int getIntParam(string name,string context=””);
   //set the full-name of the command-line arg
   string full_name = (context == “”) ?
                              name :
                              {context,”.”,name};

   //get singleton instance of uvm_commandline_processor
   uvm_commandline_processor clp = … ;

   //read the command-line
   // and store all knobs beginning with a “+” in a queue
   int retval = clp.get_arg_values(“+”,argQ);

   string valStr;
   int argVal  = <some_default_value>;

   if(retval)
     foreach(argQ[i]) begin

       //parse the string
       //NOTE: only the case of a non-empty context
       //is shown
       uvm_split_string(argQ[i],”.”,nameQ);

       //identify the sub-strings on either side
       // of the equals (“=”) sign.

      uvm_split_string(nameQ.pop_back,”=”,valueQ)

      //The string on the left-hand side
      //is the observed name
     //Try matching the names

      if(uvm_is_match(valueQ.pop_front,full_name)) begin
          //The other string in valueQ is the value
         //that is on the right hand side of “=” sign

         valStr = valueQ.pop_front;
         argVal = valStr.atoi();

       end
     end
    end

   //if no match is found, default is returned
   return argVal;

 endfunction: getIntParam
  //function to process flag type knobs
 //implementation is similar to getIntParam
 function int getBoolParam(string name,string context=”);
```

```
endclass: CommandLineProcessor
```

The PortAgentConfigInfo class has a function called setConfigParams. This will be put to use here. But first assume, that the testbench consists of an environment with two agents, along with a top-level configuration class having its own setConfigParams class. All knobs are processed whether or not they appear on the command-line. If no overriding value is present on the command-line, the CommandLineProcessor will return a default value.

The processing of the command-line happens in a nested fashion. The setConfigParams within the TopLevelConfigInfo invokes the setConfigParams function within each of the PortAgentConfigInfo instances. In this example we have two instances of PortAgentConfigInfo. The call to setConfigParams for each instance uses a unique string identifier as a context. Subsequent examples will show, how the context is also used with the knob-name on the command-line.

```
class TopLevelConfigInfo extends uvm_object;

    bit top_level_knob;
    PortAgentConfigInfo agent[2];
     .....
     function void setConfigParams(string context =””);
         foreach(agent[i])
             agent[i].setConfigParams($sformatf(“agent%0d”,i));

         //process the knob
         //no context provided,
         //since only one instance of
         //top-level configuration class is present

         top_level_knob = CommandLineProcessor::
                                    getBoolParam(“top_level_knob”);

     endfunction: setConfigParams

endclass: TopLevelConfigInfo


class PortAgentConfigInfo extends uvm_object;

     int knob;
     int addBytesWeight;
     int addWordsWeight;

   function void setConfigParams(string context = “”);
        knob = CommandLineProcessor::
                                  getIntParam(“knob”,context);
        addBytesWeight = CommandLineProcessor::
                                  getIntParam(“addBytesWeight”,context);
        addWordsWeight = CommandLineProcessor::
                                  getIntParam(“addWordsWeight”,context);
   endfunction: setConfigParams

endclass: PortAgentConfigInfo
```

Finally, the top-level configuration is set from the build_phase inside the test class. The nested parsing of the command-line is kicked off by the call to setConfigParams of the TopLevelConfigInfo instance from within the build_phase.

```
class MyTest extends uvm_test;
     .....
     TopLevelConfigInfo cfg;
     .....
     function void build_phase(uvm_phase phase);

         cfg = TopLevelConfigInfo::type_id::create("Cfg");

         //context for top-level configuration
         //could be blank in case it is not needed

        cfg.setConfigParams();
        .....

     endfunction: build_phase
     .....
endclass: MyTest
```

The following examples highlight the capabilities of CommandLineProcessor and how they are used with the simv executable.

Control “agent0”

```
simv +agent0.knob=4
```

Control “agent1”

```
simv +agent1.knob=5
```

Control all agents having this knob

```
simv +"*".knob=10
```

Control “agent0” or “agent1”

```
simv +agent0.addBytesWeight=2
simv +agent0.addWordsWeight=1
simv +agent1.addBytesWeight=3
simv +agent1.addWordsWeight=2
```

Control all knobs in “agent0” satisfying the wildcard pattern

```
simv +agent0.add”*”Weight=4
```

Control all knobs matching the pattern in all agents.

```
simv +”*”.add”*”Weight=5
```

Control a knob that takes a boolean value

```
simv +top_level_knob
```

The last case above is an example of setting a flag from the command-line. Note that when the context is not used in a configuration class, none is provided on the command-line.

6. Conclusions

This paper describes UVM agent reuse in a situation where protocol differences could set one interface apart from another. This should not detract verification engineers from reusing the same UVM agent. To achieve this, protocol level details are implemented in an abstracted UVM sequence, as illustrated in the multi-cycle protocol examples. The second example showed a recipe to deal with sending request packets in the same cycle as the write data. Use of the packet API for packing and unpacking bits while driving or monitoring signals was highlighted to show how it ties into the reuse methodology.

While the examples in sections 2 and 3.1 used parameterized interfaces, the examples could be easily reworked to not use parameterization if deemed convenient. In this case therefore, care should taken to align inputs in the interface instances to the maximum specified widths by adding dummy wires.

In one design, this methodology was used for nine protocols over thirteen separate interfaces. The protocols ranged from simple acknowledgments to more complicated two cycle address and data packets. In one instance, the parametrized agent was used to randomly drive a stall signal. The ability to easily reuse the generic port agent methodology shortened the test bench environment bring up time.

In closing, we also highlighted two other things. The first one showed how a generic UVM agent scales up in a design that has multiple communicating entities. The same agent was used in both active and passive modes on several interfaces in the same design. Finally we described some of our custom enhancements to UVM command-line processing. The enhancements allow us to specify a command-line argument at a granularity right down to the lowest level component.

We have deployed the enhanced command-line processor in our bench configuration routine to process knobs in decimals, hexadecimals, binary, boolean flags and strings. This gives us the flexibility to control stimulus based on our needs. The use of the command-line processor is not restricted to stimulus control. It can be used in any code where there is need to control several similar instances in specific ways. Also since no simulator-specific API is used, the command-line processor can be used on any simulator.

A possible future enhancement to the generic port agent would be adding interface credit management. A credit based system is used on interfaces in many designs to control the flow of traffic on an interface. The PortAgentConfigInfo class could have a configuration parameter for the number of credits available for an interface. The components in the agent could then use that information while driving or sampling the interface. This configuration parameter could be specified at run-time as an optional knob using the enhanced command-line processor. Therefore, when the knob is not used, a default value will be used, and when specified a run-time, interesting values could be exercised. A logical extension of this would then be to have a generic credit manager class inside the generic agent that could take care of the actual credit management.

7. References

[1] 1800-2012, IEEE Standard for System Verilog — Unified Hardware Design, Specification and Verification Language, IEEE 2012.

[2] Spear, Chris, Greg Tumbush, "Systemverilog For Verification: A Guide to Learning the Testbench Language Features", 3rd Edition, Springer 2012.

[3] Universal Verification Methodology (UVM) 1.2 User Guide, Accellera 2015.

[4] Pratt, Aron, "Parameterized Interfaces and Reusable VIP", Parts 1,2 and 3, Available at https://blogs.synopsys.com/vip-central/2015/01/27/parameterized-interfaces-and-reusable-vip-part-1/, https://blogs.synopsys.com/vip-central/2015/02/19/parameterized-interfaces-and-reusable-vip-part-2/, https://blogs.synopsys.com/vip-central/2015/02/24/parameterized-interfaces-and-reusable-vip-part-3/.

[5] Rich, David. and Jonathan Bromley, "Abstract BFMs Outshine Virtual Interfaces for Advanced Systemverilog Testbenches", Proceedings of DVCon 2008.

[6] UVM Cookbook, "Configuration", Available at https://verificationacademy.com/cookbook/configuration.

8. Final Notes

We are required by Oracle to let the readers know that the derivative work presented in this paper is protected by copyright. Should any questions arise, please direct them to the authors. The copyright statement is shown below: