Reusable Verification Framework

by Gunther Clasen, Ensilica

INTRODUCTION

Testbenches written in SystemVerilog and UVM face the problem of configurability and reusability between block- and system-level. Whereas reuse of UVCs from a block- to a system-level verification environment is relatively easy, the same cannot be said for the UVC's connection to the harness: The interfaces that these UVCs need changes from connections to primary inputs and outputs at block level to a set of hierarchical probes into the DUT at system level. This requires a re-write of all interface connections and hinders reuse.

This article demonstrates how to write interface connections only once and use them in both block- and system-level testbenches. The order is not important: System-level testbenches can be written without all the blocks of the DUT completed, DUT and UVM blocks can easily be interchanged. Taking care not to use virtual interfaces in the UVC but Bus Functional Models (BFM) in the interface instead – so called polymorphic interfaces, UVCs can be fully configurable as well as reusable.

REVIEW OF INTERFACES, THEIR REUSABILITY AND CONFIGURABILITY AND LIMITATIONS

During the development of a chip, it is usual to write blocks and quite often verify them stand-alone. These blocks would then typically be grouped into larger blocks / subsystems before a number of them get integrated into the complete chip. This means that the testbench needs to be scaled as the DUT grows. Modern testbenches often make use of UVM's class based verification environment. This focuses very much on reuse, mainly vertically from block level up to chip level, but also horizontally between blocks. The weak link, however, is on the one hand the connection between the static module hierarchy of the DUT (which is typically instantiated inside the testbench) and on the other hand the dynamic class-based verification environment.

Like modules, an interface is created at elaboration time, and therefore testbench writers would instantiate all interfaces which are required for a DUT in the testbench and connect them to the DUT, which is also instantiated in the testbench. Although SV interfaces were (also) meant to be a "synthesizable collection of signals" to ease the onerous connection of DUT blocks, this approach has not yet been widely adopted. This approach results in an equally onerous task of connecting the DUT in the testbench: Each signal has to be connected individually. Worse, these connections cannot be reused vertically: When the block is integrated into a subsystem, it is often desirable to keep the connection between the agent and the block and switch the agent into passive mode. This requires that the interface is connected to signals which are now buried inside the subsystem. This is not only tedious, but it may be that in post-synthesis these signals are difficult to find.

From the testbench, the SV interface is then passed into the verification environment. This could be done by either a set_vi() access function, by writing a wrapper or configuration class around the interface and passing it into the agent via UVM's configuration mechanism or, since the advent of the uvm_config_db, by passing the virtual interface directly into the uvm_config_db. This has the disadvantage that it hinders horizontal reuse, because the virtual interface type includes the type specialization, so any parameters given to the interface in the testbench must be known in the agent at compile time. Typical code would look like this:

```
interface bus_if #(int ADDR_W = 16, DATA_W = 16)
  (input logic clk,
   input logic rst_n
   );
  logic [ADDR_W-1:0] addr;
  logic [DATA_W-1:0] data;
  //...
endinterface: bus_if

module block #(int AW = 16, DW = 16)
  (input logic clk,
   input logic rst_n,
   input logic [AW-1:0] addr,
   input logic [DW-1:0] data, ... );
  ...
endmodule: block

class bus_agent #(int AW = 16, DW = 16)
                        extends uvm_component;
  virtual interface bus_if #(.ADDR_WIDTH (AW),
                                .DATA_WIDTH (DW)) vif;
  //...
endclass: bus_agent
```

```
module tb;
  bit clk, rst_n;
  bus_if #(.ADDR_W (16), .DATA_W (16)) bus_if_inst
            (.clk (clk), .rst_n (rst_n));
  block #(.AW (16), .DW (16)) DUT
    (.clk (clk),
     .rst_n (rst_n),
     .addr (bus_if_inst.addr),
     .data (bus_if_inst.data), ... );
  ...
  uvm_config_db# (virtual interface bus_if
                  #(16, 16))::set (null, "...", "vif",
                  bus_if_inst);
  ...
endmodule: tb
```

Another problem is that the agent needs to know about the virtual interface's specialization. This information can only come from the agent's own specialization, from a define command or from a package containing all configuration options unambiguously.

WRITING A REUSABLE VERIFICATION FRAMEWORK

The problem of the interface connections can be solved by changing the place where the interface is instantiated. Using the bind command to bind it into the module, the compiler treats it as if it was inside the module, giving it access to all the module's signals. By changing the interface declaration such that it uses ports instead of internal signals for the connections to the DUT, we can write the port list in the bind command in such a way that it connects to any signal in the DUT:

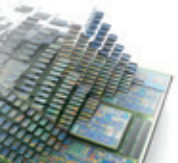
```
interface bus_if #(int ADDR_WIDTH = 16,
                   int DATA_WIDTH = 16)
  (
   input logic                  clk,
   input logic                  rst_n,
   inout  logic [ADDR_WIDTH-1:0] addr,
   inout  logic [DATA_WIDTH-1:0] wdata,
   input  logic [DATA_WIDTH-1:0] rdata,

bind small_block bus_if #(.ADDR_WIDTH (16),
                          .DATA_WIDTH (16))
                           bus_if_inst
  (
   .clk   (clk),
   .rst_n (rst_n),
   .addr  (bus1_addr),
   .wdata (bus1_wdata),
   .rdata (bus1_rdata),
  // …
   );
```

The bind command instructs the compiler to add the instantiation *bus_if_inst* of type *bus_if* to the module *small_block*. It binds it to the type *small_block*, so each instance of *small_block* will have an instance of the interface. So the signals *bus1_addr*, etc., are signals in *small_block*. It is also possible to bind the interface to a specific instance of the module only. Note that the type specializations of the module and the interface need to match, otherwise the signals which are being connected are of different width. In this instance, we want to connect the interface only to the ports of the DUT, as described in [1]. It is also recommended in [2] to use this approach for probing into the DUT, where a probe interface is bound into the DUT and connected to any internal signal. [2] also shows how to use clocking blocks and modports, which are omitted here.

Note that the direction of the port in the interface cannot be output. An output port adds a driver to the signal, which prohibits reuse of the interface in passive mode. The bind command has the same pitfalls as a "real" instantiation, like declaring signals implicitly when a connection is mistyped.

The problem of the type specialization in the agent can be avoided by eliminating the need for the agent to have a virtual interface in the first place. This can be done by using bus functional models, where a virtual base class for the BFM containing function prototypes is written which extends from uvm_component. This base class is used inside the agent and is extended in the interface, where the function prototypes are implemented. If a wrapper class is used in the same way providing a build function for the BFM, the building of the BFM can be done in the correct UVM phase and thus make it configurable via the usual UVM configuration mechanisms in the same way as any other uvm_component is configurable. [3] A function *get_api_wrapper()* must also be implemented in the interface, building an instance of the wrapper class and returning a handle on it.



```
virtual class virtual_bfm extends uvm_component;
  uvm_active_passive_enum m_active;
  // define API here, anything the agent needs to have
     access to or from
  pure virtual task wr_packet (uvm_bitstream_t l_addr,
                               uvm_bitstream_t l_data);
  pure virtual task rd_packet (uvm_bitstream_t l_addr, ref
                               uvm_bitstream_t l_data);
  pure virtual function void some_api();
  // …
endclass: virtual_bfm

interface bus_if...
  class concrete_bfm extends virtual_bfm ...
  function concrete_bfm_wrapper get_api_wrapper ...
endinterface
```

The testbench can then build the wrapper and get a handle on it by simply calling the interface's *get_api_wrapper()*.

Using Bus Functional Models also has the advantage that it enables acceleration through emulation. For that, the (timed) DUT, which runs on an emulator, needs to be strictly separated from the (untimed) verification code, which runs on a simulator. They are only allowed to communicate via transactions. [4]

Let us look at an example of a small block with a simple bus slave interface. To reuse this block both horizontally and vertically, we integrate it into a big block, together with a control block and memory. To keep it simple, the big block uses the same bus slave interface, which is passed into the control module. This uses the most significant address bit to direct the transaction to either the small block or the memory, as shown below.

This is the log when the DUT is the small block. It shows that we have an agent with a BFM configured as active, driving some data into the DUT, where it is reported when the transaction arrives. For the subsystem framework, we want to keep this agent and interface, but put it into passive mode, so that it only monitors the signals and reports ongoing activity. We need another interface and agent of the same type but with different address and data widths to connect to the bus in big_block. There is a deliberate data width mismatch in the DUT to demonstrate the use of different type specializations within the same framework (See table on the following page).

Looking at this log, we can see that we have configured the agent connecting to small_block into passive mode and the one connecting to big_block into active mode. We can further see in the log that a data transaction initiated by the BFM is seen by the monitor BUSMON in the now passive interface, before it is reported in the small_block.

CONCLUSION

Using Bus Functional Models is certainly a good way to partition a verification environment. If the BFM needs to be configurable, care must be taken to build it in the correct phase in the UVM build process and thus logically insert the BFM into UVM's hierarchy. It also enables emulation if the BFM's access functions are written according to the emulator's requirements. Using the bind command to instantiate interfaces into the DUT allows us to connect a block's ports once and reuse those connections at subsystem or chip level. Following these practices allows both horizontal and vertical reuse.

```
----------------------------------------------------------------------------------------------------------------
Name                    Type                        Size      Value
----------------------------------------------------------------------------------------------------------------
uvm_test_top            my_test                     -         @2611
  m_bus_agent           bus_agent                   -         @2709
    m_bfm               concrete_bfm                -         @2655
      m_active          uvm_active_passive_enum     1         UVM_ACTIVE
    m_bfm_wrappe        concrete_bfm_wrapper        -         @2649
    m_active            uvm_active_passive_enum     1         UVM_ACTIVE
----------------------------------------------------------------------------------------------------------------
UVM_INFO @ 15: uvm_test_top.m_bus_agent.m_bfm [some_api] m_active: 1, AW=16, DW=16
UVM_INFO @ 80: uvm_test_top.m_bus_agent.m_bfm [wr_packet] addr=1234, data=5678
UVM_INFO @ 80: reporter [small_block] BUS1 write 5678 to addr 1234
```

```
----------------------------------------------------------------------------------------------------------------
Name                    Type                    Size    Value
----------------------------------------------------------------------------------------------------------------
uvm_test_top            my_tes                  -       @2614
  m_agent_big_mod       bus_agent               -       @2767
    m_bfm               concrete_bfm            -       @2661
      m_active          uvm_active_passive_enum 1       UVM_ACTIVE
    m_bfm_wrapper       concrete_bfm_wrapper    -       @2658
    m_active            uvm_active_passive_enum 1       UVM_ACTIVE
  m_agent_sml_mod       bus_agent               -       @2726
    m_bfm                concrete_bfm           -       @2856
      m_active          uvm_active_passive_enum 1       UVM_PASSIVE
    m_bfm_wrapper       concrete_bfm_wrapper    -       @2653
    m_active            uvm_active_passive_enum 1       UVM_PASSIVE
----------------------------------------------------------------------------------------------------------------
UVM_INFO @ 10: uvm_test_top.m_agent_sm.m_bfm [some_api] m_active: 0, AW=19, DW=16
UVM_INFO @ 10: uvm_test_top.m_agent_bm.m_bfm [some_api] m_active: 1, AW=20, DW=32
UVM_INFO @ 60: uvm_test_top.m_agent_bm.m_bfm [wr_packet] addr=12345, data=56787654
UVM_INFO @ 60: uvm_test_top.m_agent_sm.m_bfm [BUSMON] write 7654 to addr 12345
UVM_INFO @ 60: reporter [small_block] BUS1 write 7654 to addr 12345
UVM_INFO @ 120: uvm_test_top.m_agent_bm.m_bfm [wr_packet] addr=87654, data=deadbeef
UVM_INFO @ 140: reporter [mem] wrote deadbeef to addr 07654
```

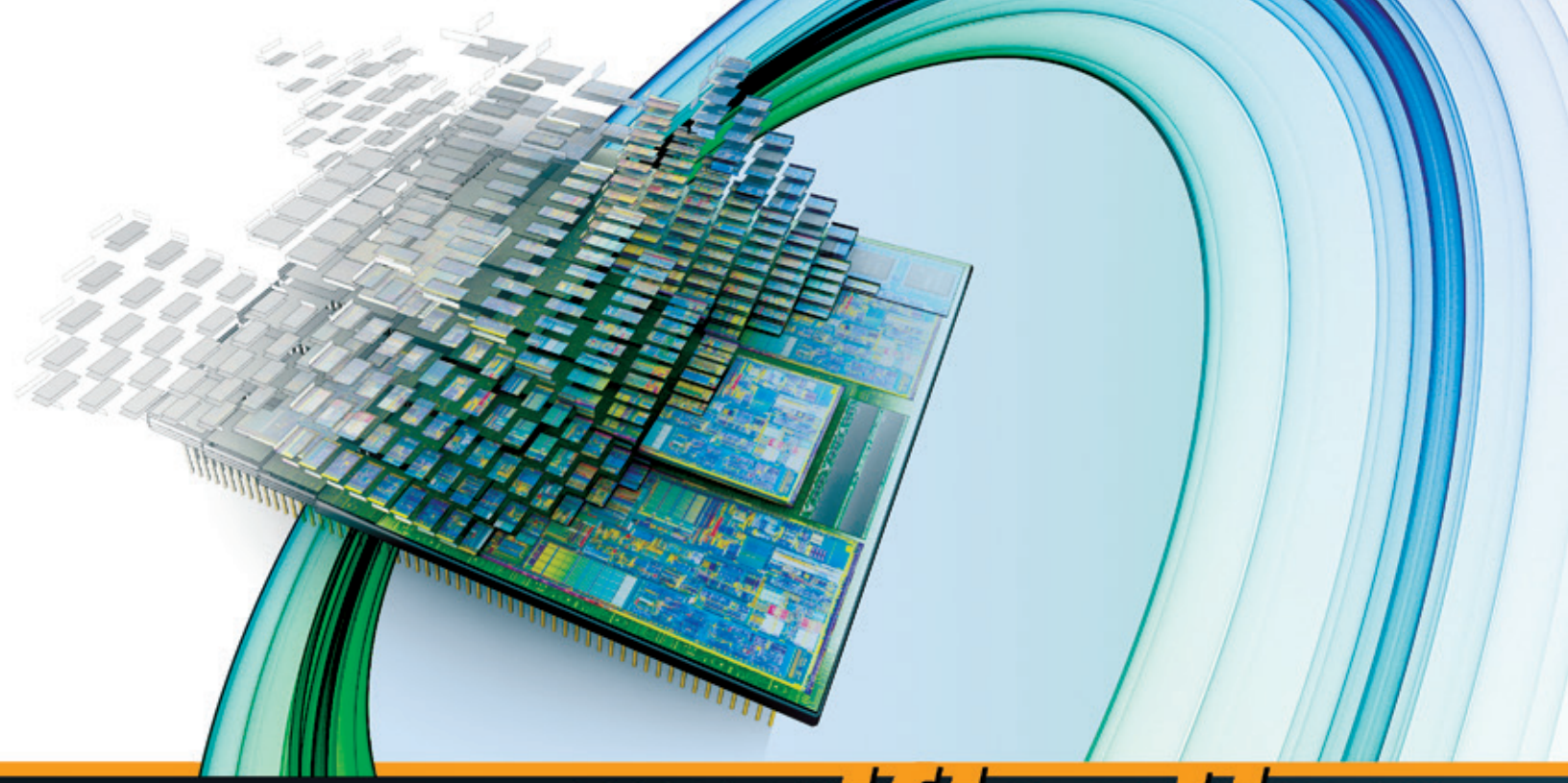
REFERENCE

[1] A proven methodology to hierarchically reuse interface connections from the block to the chip level, David Larson

[2] The Missing Link: The Testbench to DUT Connection, David Rich, 2012

[3] Flexible UVM Components: Configuring Bus Functional Models, Gunther Clasen, 2013

[4] UVM Acceleration: Creating Emulation-Ready UVM Testbenches to Boost Block-to-System Verification Productivity, Hans van der Schoot



verification HORIZONS

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

VERIFICATION HORIZONS 10 CELEBRATING TEN YEARS

Mentor Graphics®

www.mentor.com