Flexible UVM Components: Configuring Bus Functional Models

by Gunther Clasen, Ensilica

INTRODUCTION

Modern object-oriented testbenches using SystemVerilog and OVM/UVM have been using SystemVerilog interface constructs in the testbench and virtual interfaces in the class based verification structure to connect the two worlds of static modules and dynamic classes. This has certain limitations, like the use of parameterized interfaces, which are overcome by using Bus Functional Models. BFMs are now increasingly adopted in UVM testbenches, but this causes other problems, particularly for complex BFMs: They cannot be configured from the test environment, thus significantly reducing code reuse.

This article shows a way to write BFMs in such a way that they can be configured like any other UVM component using *uvm_config_db*. This allows a uniform configuration approach and eases reuse. All code examples use UVM, but work equally with the *set_config_*()* functions in OVM.

REVIEW OF VIRTUAL INTERFACES AND BUS FUNCTIONAL MODELS

A SV testbench and the DUT are static modules which are created at elaboration time and cannot be changed. Conversely, a SV class based verification environment consists of dynamic objects which are created when simulation starts, after the elaboration phase. To connect these two worlds, the use of SV interfaces in the testbench and their respective virtual interface counterparts in class components has long been propagated in the documentation of OVM/UVM and other literature [1]. SV Interfaces do indeed provide a lot of features for that very purpose: They have access to both worlds, provide modports and clocking blocks to limit access to signals and to ensure correct timing between the DUT and test environment. This document uses the term agent or (UVM) component to mean class-based objects requiring access to an interface. Examples of these are UVCs and their drivers and monitors.

An interface is instantiated in the test harness like a module and, like a module, it can be parameterized. The test environment and its components then use virtual interfaces, which are handles on the interfaces, to access signals in the testbench and ultimately drive and monitor the DUT.

To do this, the virtual interface must have the same type specialization as the interface instantiation in the testbench. In other words, any component which uses a handle on a parameterized interface must know those parameters to create the correct type specialization at elaboration time. The only way to get these parameters is by parameterizing the component class. This results in very cumbersome code and is clearly not desirable.

This problem can be overcome by using Bus Functional Models [2]: They consist of a virtual base class containing the prototype of an API which is used to communicate between the interface and the agent. In its simplest form the virtual base class may not extend from anything. However, if UVM reporting is used, it is useful to extend from *uvm_object* to avoid using the global reporter. The virtual base class is then extended inside the interface, where it has access to the interface's signals. This extended class provides the actual or concrete implementation of the BFM. The test environment needs a handle on the base class and uses that, via polymorphism, to access the API methods in the BFM. The testbench in turn creates an instance of the BFM implementation class and passes it to the test environment [3]. Extending the base class from *uvm_object* provides another advantage: This passing of the implementation class from the testbench to the test environment can then be done using *uvm_config_db*, the standard way of configuring objects in UVM.
The resulting hierarchy can be seen in Figure 1, with the arrow denoting a handle to the actual object.

```
virtual class virtual_bfm extends uvm_object;
  …
  // 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 my_if #(int BUS_WIDTH = 16);
  bit [BUS_WIDTH-1:0] addr;
  bit [BUS_WIDTH-1:0] data;
```

```
class concrete_bfm extends virtual_bfm;
  `uvm_object_utils (concrete_bfm)
  …
  task wr_packet (uvm_bitstream_t l_addr, uvm_
                  bitstream_t l_data);
    …
    data = l_data;
    addr = l_addr;
  endtask: wr_packet
endclass: concrete_bfm

function concrete_bfm get_api (string name = “”);
  static concrete_bfm api;
  if (api == null)
    api = new (name);
  return api;
endfunction: get_api

endinterface: my_if

module tb;
  my_if      if_inst_16();
  virtual_bfm vbfm_16;
  initial begin
    vbfm_16 = if_inst_16.get_api();
    uvm_config_db#(uvm_object)::set (null, “*”, “m_
                    bfm_16”, vbfm_16);
    run_test();
    ...
```

Note that it is not necessary to have an explicit handle on the *concrete_bfm* inside the interface. This can be hidden inside the *get_api()* function.

LIMITATIONS OF THIS APPROACH

A test environment would typically configure and build a number of components. In UVM, this is done in the *build_phase()* function. The *uvm_config_db#()::set()* function is used to give configuration data to the factory. The *type_id::create()* function is then used to instruct the factory to actually build a certain object, taking into account all the configuration information the factory has received. The instantiated component would typically list its configurable members in the uvm field automation macros, which hide the *uvm_config_db#()::get()* from the user. This build method is a top-down approach, building up a hierarchy

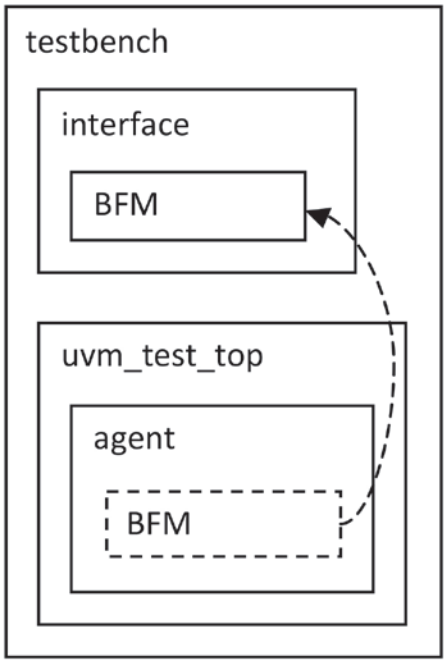


Figure 1

of the component's names. It only works because the user is required to add a call to *super.build_phase()* and in doing so ensures that each component's parent is built first. This process is started when the testbench calls *start_test()*.

Anybody writing UVM test environments will be familiar with this approach. If the BFM contains functionality which should be configured in the same way, one would expect the same approach to be used. First, the virtual base class must be extended from *uvm_component*. Next, the configurable fields must be listed in the field macros. Finally, the test environment then configures the field and builds the BFM.

```
virtual class virtual_bfm extends uvm_object;
  int m_max_burst = -1;
  …
endclass: virtual_bfm

  class concrete_bfm extends virtual_bfm;
    `uvm_object_utils_begin (concrete_bfm)
      `uvm_field_int (m_max_burst, UVM_DEFAULT)
    `uvm_object_utils_end
```

```
  …
endclass: concrete_bfm

class my_component extends uvm_component;
  …
  build_phase (uvm_phase phase);
    // configure and build “my_block”
    uvm_config_db#(uvm_bitstream_t)::set (this, “my_
                          block”, “m_num_slaves”, 3);
    my_block = my_block_t::type_id::create (“my_block”,
                                                this);
    // configure and build BFM
```

It is tempting to add configuration instructions for the BFM at this point. After all, this is how it is done for other components in the test environment. But this will not work: The BFM is built in the testbench. Any configuration data must be provided before the configured object is built, but by the time *build_phase()* executes, the BFM has already been built. This means that the testbench is the direct parent of the BFM, not the test environment. This can be seen when printing the test topology. The test environment merely has a handle on the BFM, as can be seen by the identical numbers in the “Value” column:

```
UVM_INFO @ 0: reporter [UVMTOP] UVM
        testbench topology:
------------------------------------------------------------
Name                Type          Size      Value
------------------------------------------------------------
bfm_16              concrete_bfm  -         @4755
  m_max_burst       integral      32        ‘h1
bfm_64              concrete_bfm  -         @296
  m_max_burst       integral      32        ‘h1
uvm_test_top        my_test       -         @310
  m_agent_16        my_agent      -         @5047
    m_bfm           concrete_bfm  -         @4755
      m_max_burst   integral      32        ‘h1
  m_agent_64        my_agent      -         @5069
    m_bfm           concrete_bfm  -         @296
      m_max_burst   integral      32        ‘h1
------------------------------------------------------------

UVM_INFO @ 20: bfm_16 [wr_packet] addr=
        e87a, data=26b3
UVM_INFO @ 40: bfm_64 [wr_packet addr=
        deadbeefdeadbeef, data=52c702a914f83c62
```

The only way to configure this BFM is from the testbench, like it was done in the first example.

Another consequence of the BFM being a component of the testbench is the UVM reporting output. Standard UVM report displays the full hierarchical path of the object, in this case “bfm_16” and “bfm_64”. This is correct but not helpful for debugging. It does not show which agent causes that output. It would be better if the full path was shown to where the BFM is used inside the environment.

So it is the testbench knowing what to build and the test environment knowing when to build it. How can the two be combined?

A FULLY CONFIGURABLE BUS FUNCTIONAL MODEL

For the BFM to work properly, it is mandatory that the testbench builds an object and passes it into the test environment. But does it have to be the BFM? If the testbench built a wrapper class and passed that into the test environment, it would still hide the actual path from the environment and thus maintain its reuse. The wrapper class would follow the same strategy as the BFM itself. It would have a virtual base class which is extended in the concrete implementation. Its API would simply consist of an instruction to build the BFM. This would remove the need to build the BFM in the testbench. Instead, it could be built, via the API in the wrapper, from the test environment. In doing so, the BFM would be correctly inserted into the environments hierarchy. This would guarantee that the BFM can be configured from the test environment just like any other component. The only difference is that the BFM would need to be built via the API in the wrapper instead of using *type_id::create()*. The following code demonstrates this:

```
virtual class virtual_bfm_wrapper extends uvm_object;
  function new (string name = “”);
    super.new (name);
  endfunction: new

  // API is to build the BFM
  pure virtual function virtual_bfm build_bfm
  (string name = “”, uvm_component parent = null);

endclass: virtual_bfm_wrapper
```

The wrapper base class must then be extended inside the interface into a concrete implementation:

```
class concrete_bfm_wrapper extends
virtual_bfm_wrapper;
  …
  function virtual_bfm build_bfm (string name = "", uvm_
                                  component parent = null);
    static concrete_bfm c_bfm;
    if (c_bfm == null)
      c_bfm = concrete_bfm::type_id::create (name, parent);
    return c_bfm;
  endfunction: build_bfm
  …
endclass: concrete_bfm_wrapper
```

This class is only used temporarily during construction of the test environment, so there is no immediate need to register this class with the factory. Also, in the interface, the function *get_api()* is no longer required, but instead an equivalent function is needed returning an instance of the wrapper class:

```
function concrete_bfm_wrapper get_api_wrapper
                    (string name = "");
  static concrete_bfm_wrapper api_wrapper;
  if (api_wrapper == null)
    api_wrapper = new (name);
  return api_wrapper;
endfunction: get_api_wrapper
```

Any component using this BFM can now configure it like it would configure any other component:

```
class my_agent extends uvm_component;

  virtual_bfm              m_bfm;
  virtual_bfm_wrapper m_bfm_wrapper;
  int                          m_max_burst;

  `uvm_component_utils_begin (my_agent)
    `uvm_field_object (m_bfm_wrapper, UVM_DEFAULT)
    `uvm_field_int    (m_max_burst,   UVM_DEFAULT)
  `uvm_component_utils_end

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

  function void build_phase (uvm_phase phase);
    super.build_phase (phase);
    // make sure we have BFM wrapper
    if (m_bfm_wrapper == null)
      `uvm_fatal ("CFGERR", "m_bfm_wrapper must be
                  configured")
    // configuring the BFM from here now works
    uvm_config_db#(uvm_bitstream_t)::set (this, "m_bfm",
                        "m_max_burst", m_max_burst);
    m_bfm = m_bfm_wrapper.build_bfm ("m_bfm", this);
  endfunction: build_phase

endclass: my_agent
```

The agent is now in full control over all its subcomponents and can follow good coding practices: The user of the agent only needs to configure the agent itself and does not need to be concerned about how the agent configures all its subcomponents. Figure 2 shows the resulting hierarchy.

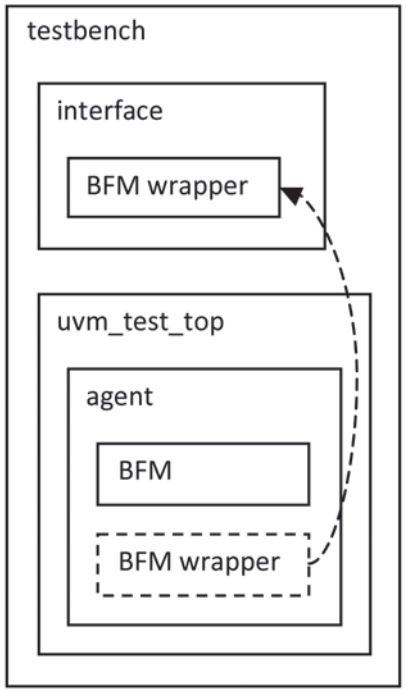


Figure 2

```
uvm_config_db#(uvm_bitstream_t)::set (this, “m_agent_16”, “m_max_burst”, 5);
uvm_config_db#(uvm_bitstream_t)::set (this, “m_agent_64”, “m_max_burst”, 8);

----------------------------------------------------------------------------------------
UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:
----------------------------------------------------------------------------------------
Name                    Type                    Size      Value
----------------------------------------------------------------------------------------
uvm_test_top            my_test                 -         @312
  m_agent_16            my_agent                -         @4937
    m_bfm               concrete_bfm            -         @335
      m_max_burst       integral                32        'h5
    m_bfm_wrapper       concrete_bfm_wrapper    -         @4766
  m_agent_64            my_agent                -         @4936
    m_bfm               concrete_bfm            -         @304
      m_max_burst       integral                32        'h8
    m_bfm_wrapper       concrete_bfm_wrapper    -         @4773
  m_bfm_16_wrapper      concrete_bfm_wrapper    -         @4766
  m_bfm_64_wrapper      concrete_bfm_wrapper    -         @4773
----------------------------------------------------------------------------------------
UVM_INFO @ 20: uvm_test_top.m_agent_16.m_bfm [wr_packet] addr=e87a, data=26b3
UVM_INFO @ 30: uvm_test_top.m_agent_16.m_bfm [rd_packet] addr=53b6, data=8d62
UVM_INFO @ 40: uvm_test_top.m_agent_64.m_bfm [wr_packet] addr=deadbeefdeadbeef, data=52c702a914f83c62
UVM_INFO @ 50: uvm_test_top.m_agent_64.m_bfm [rd_packet] addr=739b7e9a29f8c403, data=39d6ab3f093e7c25
```

Below is an excerpt from a log file which uses the code snippets in this article. The testbench instantiates two interfaces, one parameterized with 16 bit, the other with 64 bit and builds the BFM wrappers. The test environment instantiates two corresponding agents and configures their field *m_max_burst*, which is then used by the agent to configure the BFM:

It can be seen from the hierarchy that the BFM is now fully integrated into the test environment. The only component at the testbench level is *uvm_test_top*. Also, the UVM reporting now outputs the full path name of the BFM to be inside the agent, making it very easy to see which line is output by which BFM.

To maximize reuse, the virtual wrapper class can be made generic. The only BFM-specific code in it is the return value of the function. This can either be changed to *uvm_object*, or the return type can be configured via a parameter into the class. Giving that parameter a default of *uvm_object* means that the extended class may use the default specialization when used:

```
virtual class virtual_bfm_wrapper #(type T = uvm_object)
                                    extends uvm_object;
  …
  // API is to build the BFM
  pure virtual function T build_bfm (string name = “”, uvm_
                                    component parent = null);
endclass: virtual_bfm_wrapper
```

This has the advantage that it would need to be written only once and can then be added to the company's specific extension package for ease of reuse.

What each BFM must do is to implement the concrete wrapper class and function *get_api_wrapper()*. This is a very repetitive task. However, analyzing the code reveals that the only differences in the code written for different BFM are the types of both the virtual and the concrete BFM classes, i. e. *virtual_bfm* and *concrete_bfm* in the

code examples. This makes it very easy to write a macro containing class *concrete_bfm_wrapper* and function *get_api_wrapper()*, again adding them to the company's specific extension package.

CONCLUSION

Using Bus Functional Models is certainly a good way to partition a verification environment. However, the lack of configurability of the BFM is likely to catch out the novice UVM user, who often does not fully understand the exact details of the build process in UVM. This results in unnecessary debug times.

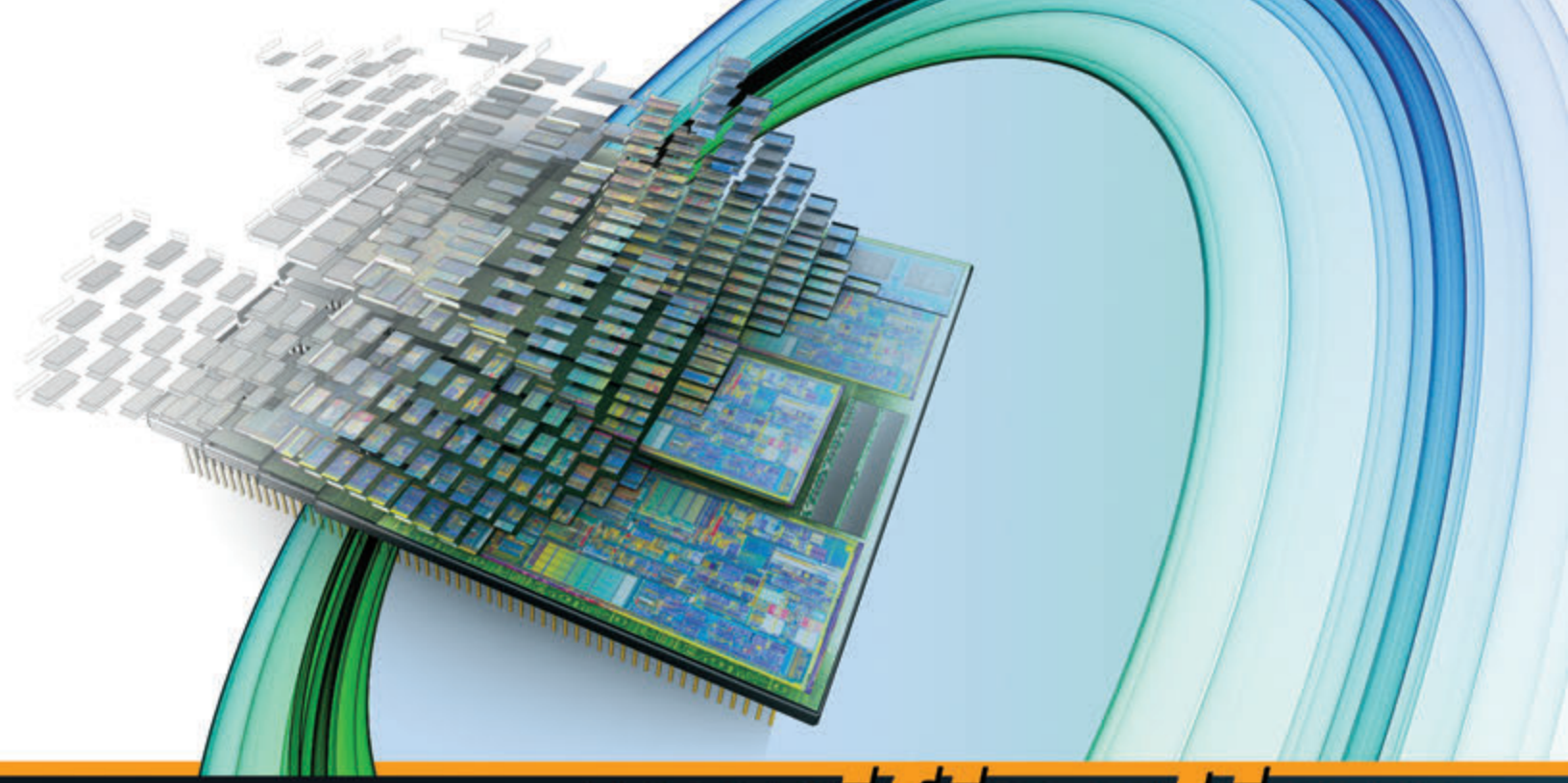
Using the approach outlined here, it has been shown that the BFM can be made to work like any other component. This increases productivity by easing the integration process. It allows to follow good coding practices by making it possible for the component to configure its own BFM. It also improves log messages. Associated with this is the small expense of having to write a generic base class for the BFM wrapper plus a concrete implementation of it for each BFM. A macro can be used for the concrete implementation, virtually eliminating the overhead of this solution.

REFERENCE

[1] Universal Verification Methodology (UVM) 1.1 User's Guide, Accellera

[2] Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches, David Rich and Jonathan Bromley

[3] Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces, Shashi Bhutada



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

Mentor Graphics®

www.mentor.com