Working within the Parameters that SystemVerilog has constrained us to

Salman Tanvir and David Crutchfield
Infineon Technologies
Lexington, KY 40507
Salman.Tanvir@infineon.com, David.Crutchfield@infineon.com

Markus Brosch
Infineon Technologies
Villach, 9500, Austria
Markus.Brosch@infineon.com

***Abstract*-Parameterized IP is a fundamental building block in System-On-Chip design. Although parameters work seamlessly with SystemVerilog modules and interfaces (HDL), the class based (HVL) side poses numerous challenges. These include issues with the HDL to HVL connection via the virtual interface, the UVM factory and parametric coverage. Various publications already exist that touch on one or another of these issues. The motivation of this work is not only to consolidate the existing knowledge on these topics but also to delve deeper into them and provide additional recommendations and solutions where possible.**

I. Introduction

Modern SoC designs are composed of a large number of IP blocks, which are typically a mix of internal and third-party vendor provided IP. In order to manage the complexity, not only is the reuse of existing IP essential, but modern platform-based design goes one step further in churning out multiple derivatives of a product by reusing a single configurable base design of those IP. Parameterization is at the heart of this reuse methodology. Parameters are well supported in the most widely used Hardware Description Languages (HDL), namely VHDL, Verilog and SystemVerilog. However, on the HVL side in SystemVerilog, the user must contend with various challenges when working with parameterized interfaces, classes and coverage. Before understanding the challenges, we must first be able to comprehend various aspects of the SystemVerilog language. Hence a short primer on the relevant features is presented first. The various challenges faced with parameters are then explained. The first issue is with the parameterized interface. The virtual interface abstraction is a useful means to access the Device Under Test (DUT) interface signals from the HVL part of the testbench. However, special handling is required for parameterized interfaces. This paper explains these challenges and presents and compares various known workarounds. The existing solutions are not able to effectively support parameterized interfaces in an emulation context which results in different VIPs being used for simulation and emulation. In the spirit of reuse, we propose a new emulation compatible solution, that does not trade off flexibility and genericity for synthesis compatibility, and hence can be used for both use cases. The next problem area highlighted is the UVM factory. Whilst working with non-parameterized classes, the UVM factory abstraction (base class & macros) is easy to use without understanding its inner workings. However, to effectively use the factory design pattern with parameterized classes, we need to dig deeper. The final challenge described is related to parametric functional coverage. Parameterized classes make parametric coverage handling very easy, but as we will see, there are drawbacks to this. Special care is required to handle parametric coverage without the use of parameterized classes. Various solutions to this problem are presented.

II. SystemVerilog Primer

A. *Background*

SystemVerilog, the predominant hardware verification language in the industry, was first released as an Acellera standard in 2002. A significant part of the language was influenced by Superlog and Vera, which in turn had links to Verilog, C/C++ and Java [1]. The interface construct was inherited from Superlog and the object-oriented programming (OOP) framework from Vera. Unlike classes, interfaces do not support dynamic polymorphism. As we will see, to solve challenges with parameterized interfaces, we must employ some tricks to make the interface pseudo-polymorphic. Our motivation to trace the origins of these SystemVerilog constructs was to understand if the complexity involved in parameter handling already existed in the precursor languages. We believe that for the large part, this complexity is unavoidable. However, we see potential for improvements with virtual interface handling that we have proposed later in this work.

B. Polymorphism

In programming languages, polymorphism is the ability to represent different types of objects via a single interface. Although there are various types of polymorphism, we shall limit ourselves to parametric and sub-type polymorphism. These can be further classified into static and dynamic polymorphism. In static polymorphism, the object type is resolved at compile time. On the other hand, dynamic polymorphism uses inheritance and virtual methods to resolve function calls at run time.

SystemVerilog parameterized classes and interfaces are examples of parametric polymorphism. As the parameters, which can themselves define a generic data type (type parameter), or be literals of various types, e.g., integrals or real are resolved at compile time, these fall under the static polymorphism category.

Each specialization (unique parameter values) of a parameterized class or interface represents a unique type. Hence a parameterized class is a generic template, and apart from the default parameters specialization, does not represent a concrete type. Concrete specializations are only established when specialized parameters are used to declare a class variable, or the type is defined via a “typedef”. Once the types have been statically defined, parameterized classes do support dynamic polymorphism.

The parameterized class type specificity has implications on sub-type (dynamic) polymorphism. In this type of polymorphism, a base class handle is used to call virtual methods overridden in the class's sub-types dynamically. To achieve dynamic polymorphism with parameterized classes, it is required to extend the parameterized class from a non-parameterized base class. This technique is exemplified in Section III-D. There is another use-case for this. SystemVerilog supports static properties and methods, which can be accessed without instantiating a class object. However, to share member variables among different specializations of a parameterized class, they must be placed in a non-parameterized base class. This is because, in SystemVerilog, each specialization of a parameterized class has a unique set of static properties.

III. The Parameterization Problem

We begin describing the parameterization problem by understanding the parameterized interface. Interfaces are typically instantiated in the top testbench HDL module or bound directly inside the DUT. Although the interface is physically instantiated on the HDL side, a handle or reference to the interface is required on the HVL side in order to drive or monitor signals. This is achieved by declaring a virtual interface variable on the HVL side. This variable defaults to a null value and must be assigned with an actual interface instance before being referenced. The UVM recommended way is to use the configuration (uvm_config_db) or resource (uvm_resource_db) databases to make this assignment [2,3]. However, the reference to the physical interface retrieved from either of the databases, must be assignment compatible to the virtual interface variable. The only allowed assignments are either the null constant, a virtual interface or an interface instance of the same type. This creates challenges for parameterized interface handling which are now discussed.

A. Parameterized Class Approach

For a parameterized virtual interface variable to point to various interface specializations, the encapsulating class component can be parameterized. On the face of it, this is a straightforward and elegant solution. However, this technique has both its pros and cons. On the plus side, multiple class specializations can easily be created in a single environment to handle different parameters. Furthermore, this technique supports parametric functional coverage out of the box, and as we shall see in Section III-E, without this, specific workarounds are required in place of it.

The first drawback of parameterized class components is the parameter ripple effect [4-6]. A parameterized agent requires a parameterized UVC environment, which in turn requires a parameterized testbench environment, if the testbench is intended to be reused. There are implications to using parameterized classes, and due to this ripple effect, these cannot be isolated in one place.

Before explaining these implications, it must be noted that parameterized classes are very effective if used correctly. The ideal use case for these is as base classes, as exemplified in various UVM classes. The register adapter and predictor classes allow type overriding for the bus transaction type. Similarly, the register front-door read and write sequence types are configurable. The base classes are typically overridden with a specific type and integrated in a testbench as a non-parameterizable specialization. The same is true for the UVM driver, sequencer and monitor.

The main drawback for parameterized classes is that the UVM utility macros only support the type and not name based factory. This is because the macros neither declare the string type name, nor the "get" accessor method for it. Although this is not complicated to fix manually, it can be unclear for the novice user. The string name is not required when overriding by type, but it makes debugging more difficult. Consider the type overrides in the following figure.

```
set_inst_override_by_type("...master_drv_proxy",...driver_bfm_proxy_abstract::get_type(), ...master_driver_bfm_proxy16::get_type());
set_inst_override_by_type("...master_drv_proxy",...driver_bfm_proxy_abstract::get_type(), ...master_driver_bfm_proxy32::get_type());
```

Figure 1. Factory type override example

As can be seen in the factory.print() debug method output below, the override type is not printed correctly.

```
#### Factory Configuration (*)

Instance Overrides:

Requested Type                              Override Path                                       Override Type
------------------------------------------  --------------------------------------------------  -------------
dummy_uvc_master_driver_bfm_proxy_abstract  ...dummy_env.agents0.master_driver.master_drv_proxy  <unknown>
dummy_uvc_master_driver_bfm_proxy_abstract  ...dummy_env.agents2.master_driver.master_drv_proxy  <unknown>
```

Figure 2. Factory debug with parameterized classes.

The missing string type name also has implications outside the factory. The type name for a uvm_component instance is typically used to tag debug messages e.g. with the ID argument of the `uvm_info macro. The get_type_name() is used for this, and due to the missing type name string, it returns "<unknown>".

```
`uvm_warning(this.get_type_name(), $sformatf("[REG_FIELD_READ] Register: %s | Field: %s not found", reg_name, field_name))

Output: @ 1: reporter [<unknown>] [REG_FIELD_READ] Register: config | Field: intr_en not found
```

Figure 3. Using Get_type_name() with parameterized classes.

```
class driver_bfm_proxy#(int unsigned addr_width = 16,int unsigned data_width = 16)
  extends driver_bfm_proxy_abstract;

  virtual hdl_drv_bfm#(addr_width ,data_width) v_bfm;

  rand logic[addr_width-1:0] addr;
  rand logic[data_width-1:0] data;

  //`uvm_object_param_utils_begin(driver_bfm_proxy#(addr_width, data_width))
  //  `uvm_field_int(addr, UVM_ALL_ON)
  //  `uvm_field_int(data, UVM_ALL_ON)
  //`uvm_object_utils_end

  localparam type_name = $sformatf("driver_bfm_proxy_%0d_%0d",addr_width,data_width);
  virtual function string get_type_name();
    return type_name;
  endfunction

  typedef uvm_object_registry #(driver_bfm_proxy#(addr_width,data_width),type_name) type_id;
  static function type_id get_type();
    return type_id::get();
  endfunction
```

Figure 4. Parameterized classes registration fix example

In the code snippet in Figure 4, the uvm component utility macro for paramterized classes has been commented out, and in addition to the standard registration code, the string type name and corresponding get accessor method have been added. With this addition, not only is overriding by string name possible, but debugging the factory is much easier when overriding by type.

B. Maximum Footprint Approach

The maximum footprint approach defines signals with a maximum width to cover all desired signal sizes. The maximum size can be defined using a macro to allow resizing if required. A clear advantage of this approach is that different signal widths can be supported without the need of parameterizing the interface and class-based components, hence avoiding the parameter ripple effect and UVM factory issues. The disadvantages are the overhead to specify which part of the interface connects to the DUT and to tie off the unused part of the bus [6]. Additionally, if a large part of the signal is not utilized, debugging becomes more difficult.

C. Polymorphic Interface

The polymorphic interface approach realizes the testbench to DUT connection via APIs rather than direct signal referencing. Instead of a specific virtual interface, an abstract class handle is used to reference any concrete class that extends from it using OOP polymorphism. The abstract class defines an API to pass information to and from the BFM. The concrete class implements the accessor/mutator methods for the interface and resides in the HDL BFMs. As shown in Figure 5, these APIs interact with those defined in the BFM in order to drive and monitor the interface signals.

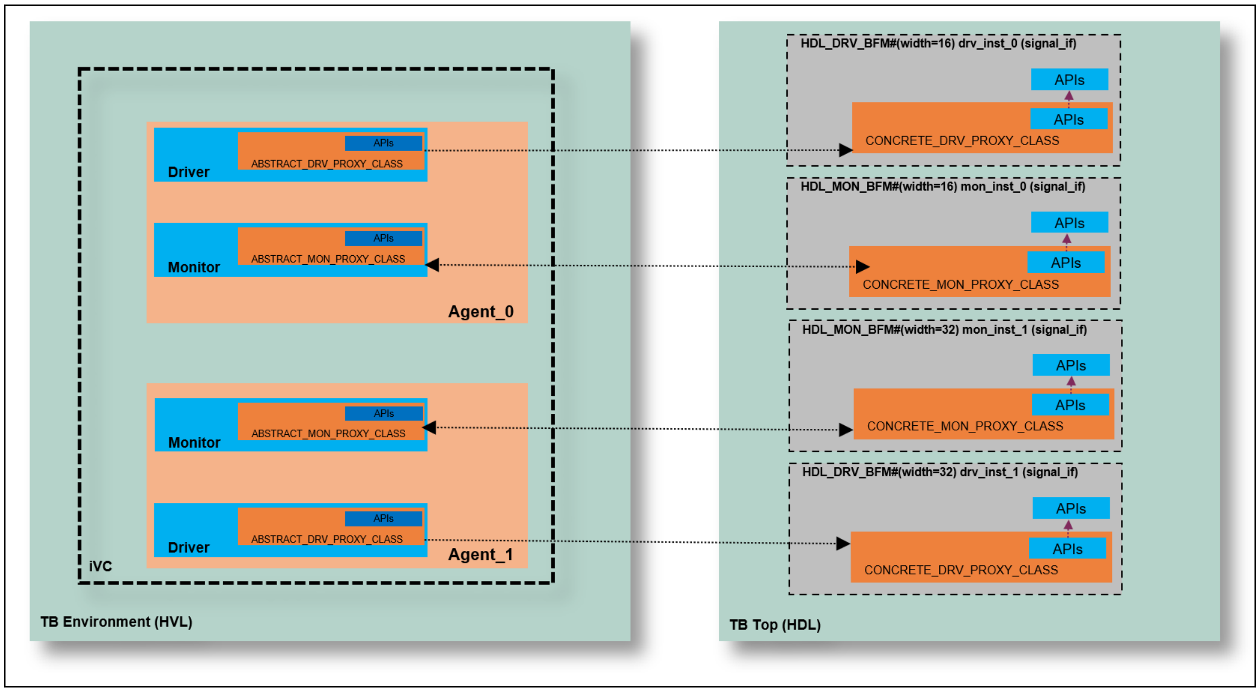


Figure 5. Polymorphic Interface

A clear advantage of this approach is the decoupling of the class-based side from the SystemVerilog interface which results in a high flexibility. The generic HVL API can be used to connect to different interface types, which can not only be different specializations of a parameterized interface, but also varying flavors of a protocol. As the concrete class is scoped inside the BFM, it can upwards reference the BFM parameters without itself being parameterized. This means we are immune to the parameter ripple effect as well as the UVM factory issues described previously.

The disadvantages of this approach are emulation incompatibility as classes in BFMs are not synthesizable and that all members of the interface need to be accessed via the accessor/mutator methods in the API and never by direct reference [7].

D. Emulation Compatible Polymorphic Interface

The polymorphic interface approach requires the concrete class to be built inside the BFM on the HDL side. As synthesizable code is a requirement for emulation, this is a problem. This paper proposes a new flavor of the polymorphic interface that removes the concrete class object from the BFM, hence making it emulation compatible.

In this approach, the concrete class is kept solely on the HVL side. The first consequence of this relocation is that the concrete proxy class must be parameterized. In the original polymorphic interface approach, the proxy class is scoped within the parameterized interface and hence does not need to be parameterized itself. As the concrete class is no longer inside the interface, it uses a virtual interface handle to communicate with it. The setting of this virtual handle requires some steps which are described at the end of this section. The abstract base class handle within verification component classes remains parameter free, as otherwise, we cannot use polymorphism to point to multiple concrete class specializations with a single interface.

```
class driver_bfm_proxy#(int unsigned addr_width = 16,int unsigned data_width = 16)
  extends driver_bfm_proxy_abstract;

  virtual hdl_drv_bfm#(addr_width ,data_width) v_bfm;

  rand logic[addr_width-1:0] addr;
  rand logic[data_width-1:0] data;

  virtual task automatic do_drive(drv_item tr_arg );
    drv_item #(addr_width,data_width) tr;
    if (!$cast(tr,tr_arg))
      `uvm_fatal("FATAL CAST", "Incompatible types for cast!");
    v_bfm.do_drive(tr.status,tr.write,tr.addr,tr.data);
  endtask
```

Figure 6.Parameterized concrete class with virtual interface handle.

Now that the concrete class is parameterized, we do have to deal with the UVM factory issues that were described in Section III-A. However, we do not need to parameterize the verification component environment class itself, and hence are immune to the parameter ripple effect. The second consequence of descoping the proxy class from the parameterized interface is that the top testbench environment must now build each specialized proxy class instance and set the corresponding verification component abstract class handles. The UVM factory is an ideal candidate to automate these steps. However, we would like to highlight that registering abstract classes with the factory is not possible in UVM 1.2. For those who are unable to migrate to the latest version, a brief explanation of the limitation and some workarounds are presented here. The root of the problem lies inside a method in the UVM component and object registry classes [9]. As can be seen in the create_component() method for UVM 1.2 in Figure 7, the constructor of the class to be registered is called. The corresponding method for UVM objects is similar.

```
// uvm_component_registery class (UVM-1.2)
virtual function uvm_component create_component (string name,
  uvm_component parent);
  T obj;
  obj = new(name, parent);
  return obj;
endfunction

//uvm_abstract_component_registry class (UVM 2020-2.0)
virtual function uvm_component create_component (string name,
  uvm_component parent);
  `uvm_error(
    "UVM/ABST_RGTRY/CREATE_ABSTRACT_CMPNT",
    $sformatf( "Cannot create an instance of abstract class %s (with name %s and parent %s).
    Check for missing factory overrides for %s.", this.get_type_name(),
    name, parent.get_full_name(), this.get_type_name() )
  )
  return null;
endfunction
```

Figure 7. Abstract class registration class.

As virtual class construction is not allowed by the UVM LRM, this results in a compile error by most EDA tools. In UVM 2020-2.0, a dedicated registration class is introduced for abstract components where the create_component() method does not construct any components but raises an error if an abstract class override is unsuccessful.

We can recommend two workarounds to this problem for UVM 1.2. The simplest solution is to make the abstract class non-virtual. Although the base class does not provide any functionality and simply serves as a generic interface, as a non-virtual class cannot contain pure virtual functions, the API methods need to be implemented. As these base APIs should not be called directly by the user, these can be embedded with an error message. The only other case when these can trigger, is when the factory override is not specified correctly. The occurrence of such errors can be mitigating by generating the overrides.

```
class driver_bfm_proxy_abstract extends uvm_object;

  `uvm_object_utils(driver_bfm_proxy_abstract)

  virtual function void error_handler(string tag = "error_handler" );
      `uvm_fatal({"PURE_BASE_CLASS: ",tag}, "This API should not be called directly! Please fix factory override");
  endfunction

  virtual task automatic do_drive(drv_item tr_arg);
    error_handler("do_drive");
  endtask
```

Figure 8. Non-virtual base proxy class.

An alternate workaround to the factory abstract registration limitation is to keep the abstract base class virtual, and instead of using the factory for overriding, to manually create the required specialized concrete proxies in the base test and assign the abstract class proxy handles. We lose the automation of the factory, but these steps can also be automated with a generator. As the required specialized types are explicitly instantiated in the testbench, this can be simpler for junior engineers to understand. Furthermore, unlike the factory-based approach described below, no type casts are required to set the virtual interface handle in the proxy. For comparison, a short example of this simplified version is presented after the factory-based approach.

```
class uvc_env extends uvm_env;
  uvc_agent agents[];
endclass: uvc_env

class tb_env extends uvm_env;
  uvc_env uvc_env;
endclass: tb_env

typedef driver_bfm_proxy# (16, 16) driver_bfm_proxy16;
typedef driver_bfm_proxy# (32, 32) driver_bfm_proxy32;

class tb_base_test extends uvm_test;
  tb_env tb_env;

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

// Factory overrides
set_inst_override_by_type("tb_env.uvc_env.agents0.driver.drv_proxy",
  driver_bfm_proxy_abstract::get_type(), driver_bfm_proxy16::get_type() );
set_inst_override_by_type("tb_env.uvc_env.agents2.driver.drv_proxy",
  driver_bfm_proxy_abstract::get_type(), driver_bfm_proxy32::get_type() );

  ...
```

Figure 9. Factory override example.

To use the factory to create the concrete proxies, the instance-based factory override methods must be used. This is because multiple specialized instances of the class component can co-exist in an environment. This is illustrated in Figure 9. In this example, a verification component environment containing an array of agents is being integrated into a testbench. Before specifying the instance overrides for the factory to create the required parameterized concrete proxy classes, the specialized types are defined using typedefs. The factory instance override method is then used to override the abstract proxy handle within each agent as required.

Now that the concrete proxy handles have been assigned, the encapsulated virtual interface handle is still null. Setting this handle involves more steps in the factory approach and is simpler when creating the concrete proxy directly in the base test. In the latter case, we have class objects of the specialized type and can directly set the virtual interface handle via the configuration database. In the factory approach however, as the agents hold a handle to the abstract proxy type, a dynamic cast is required to reference the virtual interface handle. However, these steps can be abstracted away and automated in a generic way, as illustrated in the static method in Figure 10.

```
class set_vif_util#(type V_BFM_T=int , type PROXY_T=int, type ABSTRACT_PROXY_T=int);
  static PROXY_T proxy_handle;
  static V_BFM_T vif_handle;
  static function bit set_vif( ABSTRACT_PROXY_T abstract_proxy,string scope = "", name ="");

    if (!$cast(proxy_handle,abstract_proxy))
      `uvm_fatal("FATAL CAST", "Incompatible types for cast!");

    if(!uvm_resource_db #(V_BFM_T)::read_by_name(scope, name , vif_handle))
      `uvm_fatal("FATAL V_BFM", "Virtual BFM config_db retrieval failed!");

    proxy_handle.v_bfm=vif_handle;

  endfunction

endclass
```

Figure 10. Generic virtual interface setting utility method.

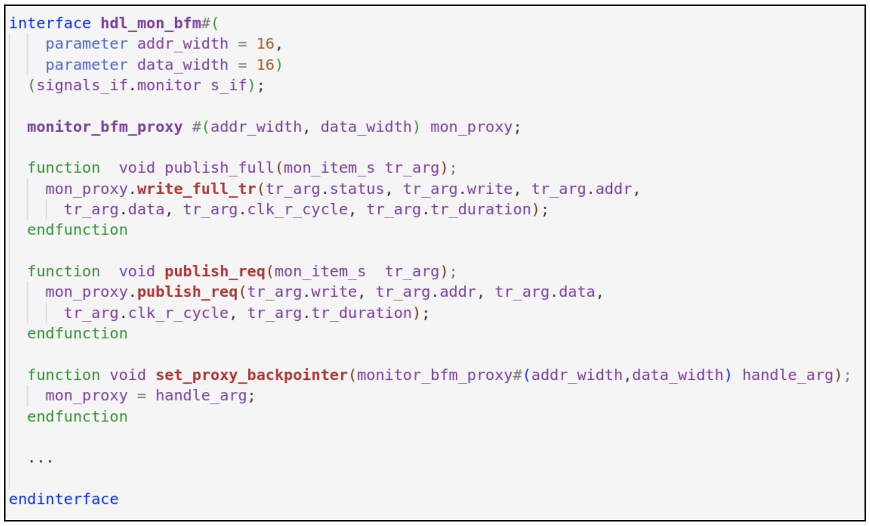
As all types including the abstract and concrete proxy, as well as the virtual interface, are parameterized, this utility function can be reused in any UVC that uses this architecture. The virtual interface pointers can be set in the connect phase as shown in the following figure.

```
module hdl_tb_top;

  initial begin
    uvm_resource_db #(virtual hdl_drv_bfm#(16,16))::set("*", "drv0",drv0);
    uvm_resource_db #(virtual hdl_drv_bfm#(32,32))::set("*", "drv2",drv2);
  end
endmodule

function void tb_base_test::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

    set_vif_util#(virtual hdl_master_drv_bfm# (16,16),driver_bfm_proxy16,
      driver_bfm_proxy_abstract)::set_vif(uvc_env.agent[0].drv_proxy,"*","drv0");
    set_vif_util#(virtual hdl_master_drv_bfm# (32,32),driver_bfm_proxy32,
      driver_bfm_proxy_abstract)::set_vif(uvc_env.agent[2].drv_proxy,"*","drv2");

...
endfunction
```

Figure 11. Virtual interface setting example.

Up till this point we have only shown the driver, as the monitor is identical. The one aspect where it does differ though, is that the HDL monitor BFM must communicate in the reverse direction with the HVL side. Although classes cannot be constructed within an emulator, class objects that have been constructed on the HVL side can be referenced. Emulators allow the setting of such class handles via function calls. This is illustrated in Figure 12. The

HDL BFM monitor passes a struct to the proxy, which converts it into a class transaction item and hands it over to the HVL monitor for publishing.



```
interface hdl_mon_bfm#(
    parameter addr_width = 16,
    parameter data_width = 16)
  (signals_if.monitor s_if);

  monitor_bfm_proxy #(addr_width, data_width) mon_proxy;

  function  void publish_full(mon_item_s tr_arg);
    mon_proxy.write_full_tr(tr_arg.status, tr_arg.write, tr_arg.addr,
      tr_arg.data, tr_arg.clk_r_cycle, tr_arg.tr_duration);
  endfunction

  function  void publish_req(mon_item_s  tr_arg);
    mon_proxy.publish_req(tr_arg.write, tr_arg.addr, tr_arg.data,
      tr_arg.clk_r_cycle, tr_arg.tr_duration);
  endfunction

  function void set_proxy_backpointer(monitor_bfm_proxy#(addr_width,data_width) handle_arg);
    mon_proxy = handle_arg;
  endfunction

  ...

endinterface
```

Figure 12. Monitor BFM

Figure 13 illustrates the complete emulation compatible polymorphic interface architecture.

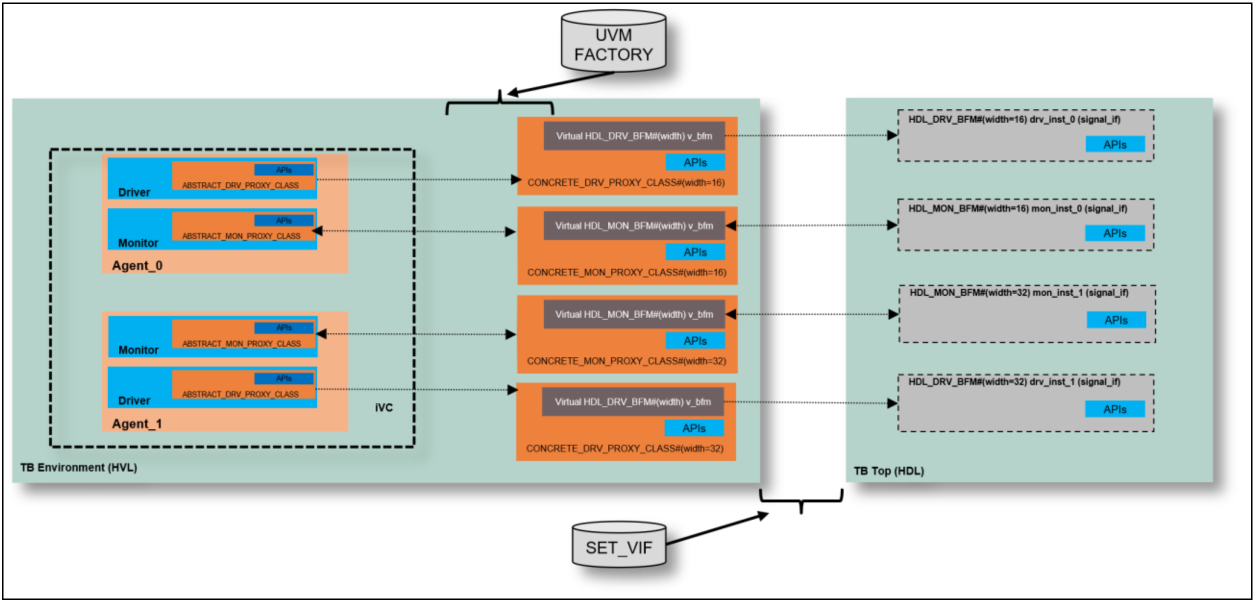

UVM FACTORY
Virtual HDL_DRV_BFM#(width) v_bfm
APIs
CONCRETE_DRV_PROXY_CLASS#(width=16)
Virtual HDL_MON_BFM#(width) v_bfm
APIs
CONCRETE_MON_PROXY_CLASS#(width=16)
Virtual HDL_MON_BFM#(width) v_bfm
APIs
CONCRETE_MON_PROXY_CLASS#(width=32)
Virtual HDL_DRV_BFM#(width) v_bfm
APIs
CONCRETE_DRV_PROXY_CLASS#(width=32)
APIs
ABSTRACT_DRV_PROXY_CLASS
Driver
APIs
ABSTRACT_MON_PROXY_CLASS
Monitor
Agent_0
APIs
ABSTRACT_MON_PROXY_CLASS
Monitor
APIs
ABSTRACT_DRV_PROXY_CLASS
Driver
Agent_1
iVC
TB Environment (HVL)
HDL_DRV_BFM#(width=16) drv_inst_0 (signal_if)
APIs
HDL_MON_BFM#(width=16) mon_inst_0 (signal_if)
APIs
HDL_MON_BFM#(width=32) mon_inst_1 (signal_if)
APIs
HDL_DRV_BFM#(width=32) drv_inst_1 (signal_if)
APIs
TB Top (HDL)
SET_VIF

Figure 13. Emulation compatible polymorphic interface.

For comparison, the simplified factory-free approach is exemplified in Figure 14. The concrete proxy class instance is created in the HVL top testbench and the virtual BFM handle is set before registering the proxy with the uvm_resource_db. No factory overrides are required as the abstract proxy handles can be set by retrieving the required concrete specializations from the uvm_resource_db.

```
module hvl_tb_top;
  master_driver_bfm_proxy#(16,16) concrete_proxy;

  signals_if#(16,16)         s_if();
  hdl_master_drv_bfm#(16,16) master_drv(s_if);

  initial begin
    concrete_proxy = new();
    concrete_proxy.v_bfm = master_drv;
    uvm_resource_db #(master_driver_bfm_proxy_abstract)::set("*", "master_drv",concrete_proxy);
    run_test();
  end
endmodule

function void tb_base_test::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  if(!uvm_resource_db #(master_driver_bfm_proxy_abstract)::read_by_name("",
     "master_drv", tb_env.uvc_env.agents[0].master_driver.master_drv_proxy))
    `uvm_fatal("FATAL V_BFM", "Virtual BFM config_db retrieval failed!");

...
endfunction
```

Figure 14. Emulation compatible polymorphic interface without using the factory.

As can be seen in Figure 13, the interface to be monitored is passed to the monitor BFM as a port argument. Similarly, it is also passed to the driver BFM. However, if a parameterized interface is passed to a BFM interface, most EDA tools flag this as a warning due to the following IEEE-1800-2017 specification: *Although an interface may contain hierarchical references to objects outside its body or ports that reference other interfaces, it shall be illegal to use an interface containing those references in the declaration of a virtual interface [12]*. Although this style of parameterized interface passing through interface ports has worked for us despite the warning, we request the SystemVerilog working group to revise this pessimistic restriction, and specifically clarify which scenarios should not be allowed. To be future proof, we have restructured the connection of the interface by using the harness approach instead [5]. This technique is exemplified in Figure 15. The BFMs and interfaces are encapsulated in a harness that gets bound into the DUT. Instead of passing the interface into the BFM ports, we rely on SystemVerilog's upwards referencing resolution. As the harness gets bound into each target specialized instance, the BFMs can reference the adjacent interface.

```
interface harness#(int addr_width = 16, data_width = 16);

  signals_if dut_if(.data(dut.data), .addr(dut.addr)):
  hdl_mon_bfm#(addr_width , data_width) mon_bfm_inst();
  hdl_drv_bfm#(addr_width , data_width) drv_bfm_inst();

endinterface

bind dut harnesss(.addr_width(addr_width), .data_width(data_width)) harness_inst();
```

Figure 15. Harness approach to avoid passing interface through BFM ports.

Figure 16 illustrates the harness concept. The HVL to HDL connection is the same as shown in Figure 13.

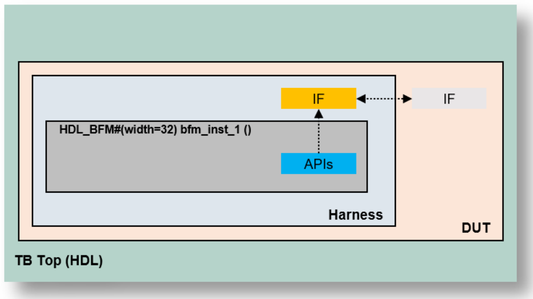


Figure 16. Bound harness

E. Parameterized Functional Coverage

As mentioned in Section III-A, coverage defined as a parameterized class does not require any special handling. However, to avoid the inherent parameter ripple effect, alternative solutions are recommended. As we have seen, handling parameters comes with its challenges, and should be avoided if possible. As functional coverage is built and sampled on the HVL side, our first recommendation is to convert static design parameters to dynamic configuration fields and pass these as arguments to the constructor of generic cover groups [11]. However, if we want to pass fields from a configuration object, we must deal with the configuration phasing problem [10]. There is a restriction on the creation of the cover group, which can only be created in the constructor of the encapsulating class. For encapsulating classes of the uvm_component type, the configuration object is only available after the calling of the new constructor and hence cannot be used as cover group generics. As uvm_objects can be constructed anytime, one solution could be to encapsulate cover groups inside objects instead of components and call the new constructor after the configuration is available. The problem here is that the new constructor prototype is fixed for uvm_objects and cannot be extended to pass in cover group generics. This can be solved by encapsulating the cover group in an arbitrary class instead of uvm_object, where the constructor can be prototyped at will. Another solution is to pull the configuration object in the constructor via a redundant call to the configuration database. An alternative method that uses parameters, similar to the strategy employed in the emulation compatible polymorphic interface from the previous section, is to use the UVM factory to override a non-parameterized base coverage class with the desired parameter specialized child class [10].

IV. Conclusion

Parametric IP design is a fundamental technique in modern SoC development. Although developing and integrating such IP using SystemVerilog is straightforward, parameter handling on the class-based side is complex and introduces significant challenges for verification.

The virtual interface is the fundamental link between the HDL and HVL sides, and although it works smoothly without parameters, its non-polymorphic nature makes parameterized virtual interface handling difficult. A comparison of the various existing solutions to this problem is presented in this paper. Although each approach has its pros and cons, none of them effectively supports the emulation use case. In our experience, we have either seen dedicated VIPs being used for simulation and emulation, or a common one that greatly sacrifices genericity for synthesizability. We have extended the polymorphic approach to solve this problem. The presented approach has not only been successfully applied to emulation compatible parameterized VIP in our company, but it has also enabled us to provide a single generic sideband UVC for both simulation and emulation [13].

Finally, we have discussed the effect of parameterization on functional coverage. Again, there are various solutions, each with its own benefits and drawbacks, but for us the polymorphic approach that uses the factory stands out for its simplicity.

With this work we have covered various challenges that we faced using SystemVerilog parameters and hope that it guides others who find themselves in a similar situation.

REFERENCES

[1] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz and Simon Davidmann, “Verilog HDL and its ancestors and descendants”, Proceedings of the ACM on Programming Languages, Volume 4, Issue HOPL, Article No.: 87, pp 1–90, June 2020.
[2] Siemens EDA, Universal Verification Methodology UVM Cookbook, Interfaces and virtual interfaces.
[3] The Untapped Power of UVM Resources and Why Engineers Should Use the uvm_resource_db API by Clifford E.Cummings, Heath Chambers and Mark Glasser, DVCon US, 2023.
[4] Pondering Parameterization, March 2020, Verilab.
[5] Advanced UVM Tutorial Taking Reuse to the Next Level (Parameterized Classes), Dvcon, Europe, 2015.
[6] Parameterize Like a Pro, Verilab, DVCon-US 2020
[7] The Missing Link: The Testbench to DUT Connection by Dave Rich, DVCon US, 2012
[8] Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces by Shashi Bhutada, Mentor Graphics
[9] Registering Abstract Classes with the UVM Factory, Tudor Timisescu, Verification Gentleman.
[10] Effective SystemVerilog Functional Coverage: design and coding recommendations, Verilab
[11] IEEE Std 1800-2017 standard for SystemVerilog. Chapter 19.5.1.
[12] IEEE Std 1800-2017 standard for SystemVerilog. Chapter 25.9.
[13] A Generic Approach to Handling Sideband Signals by Markus Brosch and Salman Tanvir, DVCON Europe, 2019.