2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Testbench flexiblity as a foundation for success

Verification coding methodology to achieve a flexible testbench, increase code reuse, enable parallel verification execution and ease of integration

Ana Sanz Carretero, Xilinx, Dublin, Ireland (*asanzcar@xilinx.com*)

Katherine Garden, Xilinx, Edinburgh, United Kingdom (*kgarden@xilinx.com*)

Wei Wei Cheong, Xilinx, Edinburgh, United Kingdom (*weiweic@xilinx.com*)

Abstract— **This paper presents a flexible verification architecture and coding methodology that allows a single UVM testbench to adapt and support all levels of integration (unit-level, top-level, system level, encrypted simulation model level, etc.), maximizing code reuse and de-risking any potential last-minute DUT top-level architectural changes. It targets multi-block designs where the specific number of block instances, type of block instances and their dependencies may vary in the future. This methodology has strong foundations in Object Oriented Programming (OOP) and leverages some of its techniques for verification to increase testbench code reuse, facilitate integration, unify and standardize development, facilitate parallel verification execution, as well as reducing the overall verification time.**

Keywords— UVM, SystemVerilog, flexibility, reuse, integration, OOP, methodology, testbench, environment, configuration, constrained random testing, directed testing, Device Under Test (DUT)

I. Introduction

A diverse ecosystem of customer applications in the market demands highly adaptable designs that are power and cost-efficient. Xilinx Zynq RFSoC DFE [1] provides an adaptive combination of soft and hard IP that can evolve with 5G standards and provide flexibility to customers for a diverse range of use cases. Flexible hardware designs require a flexible verification architecture to ensure that the testbench can adapt quickly to potential architectural changes as blocks develop and through future device generations.

Project cycles have become shorter over time to remain competitive in the market, leaving verification with very aggressive schedules. Enabling parallel verification development while reducing integration effort, maximizing code reuse and reducing testbench debug time have become critical to achieve first-time right silicon in a short project cycle.

This paper presents a UVM verification architecture and methodology with strong foundations in OOP that was used to successfully verify a multi-block design similar to Figure 1 top-level diagram. This design is composed of a number N of IPs that we will refer to as blocks, e.g. BLOCK A. These blocks are grouped together in different configurations to form different Hard IPs, e.g. “HardIP 1” contains 2 instances of BLOCK A IP and 1 of BLOCK B IP. This is what we refer to as top-level verification. These blocks can be chained together to form a final system to meet the customer needs. This is what we refer to as system-level verification.

By using this architecture and methodology, we achieved:

- A flexible testbench that allowed us to support:
    - Unit-level DUTs (for all different Block types),
    - Top-level DUTs (for all different Hard IP types),
    - System-level DUTs for a configurable system type
- Maintainable and reusable coding that facilitated vertical and horizontal reuse
- Reduced testbench debug
- Maximal reuse of tests at the different integration levels

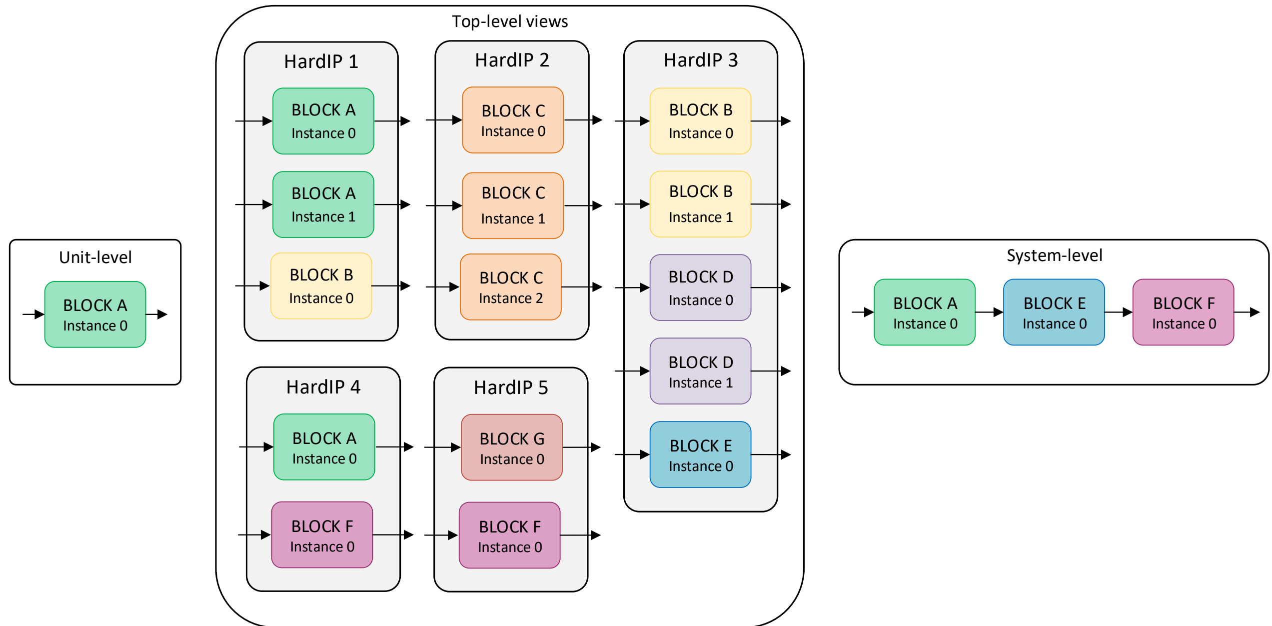

Figure 1 Examples of multi-block design levels of integration for verification

II. UNIFIED TESTBENCH METHODOLOGY FOR REUSE AND INTEGRATION

A. *Flexible testbench architecture*

In order to achieve a flexible testbench architecture that supports unit-level, top-level, and system-level verification, as well as other required internal integration levels, it is critical to have a testbench that can adapt to match the final DUT.

1) Makefile and compilation

We implemented different compilation targets as part of the Makefile to enable selective and independent DUT compilation. By using separate DUT file lists at the different integration levels and compile time defines, we were able to provide the flexibility required. See Makefile sample code in Figure 2.

```
NUM_BLOCKA ?= 0
NUM_BLOCKB ?= 0
#...
ifeq ($(TOPOLOGY), BLOCKA)
  NUM_BLOCKA := 1
  DEFINES     += +define+TB_BLOCKA
endif
# Other blocks
ifeq ($(TOPOLOGY), TOP_HARDIP1)
  NUM_BLOCKA := 2
  NUM_BLOCKB := 1
  DEFINES     += +define+TB_TOP_HARDIP1
endif
# Other top-level or system level views
```

```
#------------------------------------------------
# Block Makefile variables
#------------------------------------------------
ifneq ($(NUM_BLOCKA), 0)
  BLOCK_INCDIR    += +incdir+$(VERIF)/env/blocks/blocka
  BLOCK_PKGS      += $(VERIF)/env/blocks/blocka/blocka_env_pkg.sv
  BLOCK_IFS       += $(VERIF)/env/blocks/blocka/blocka_vif.sv
  BLOCK_DUT_WRAP  += $(VERIF)/tb/blocka_dut_wrapper.sv
  BLOCK_DEFINES   += +define+TB_NUM_BLOCKA=$(NUM_BLOCKA)
endif
#...
```

Figure 2 Makefile topologies, defines and include implementation sample code

2) Testbench top

The testbench top instantiates the selected DUT(s) by using compile time defines from the Makefile. It also generates the appropriate modules that we refer to as *DUT wrappers* that hold the DUT interfaces and take care of the interface connections and interface encapsulation into a virtual interface bundle. This interface bundle is passed from the **tb_top** to the top-level environment. Refer to Figure 3 for **tb_top** diagram and code snippet.

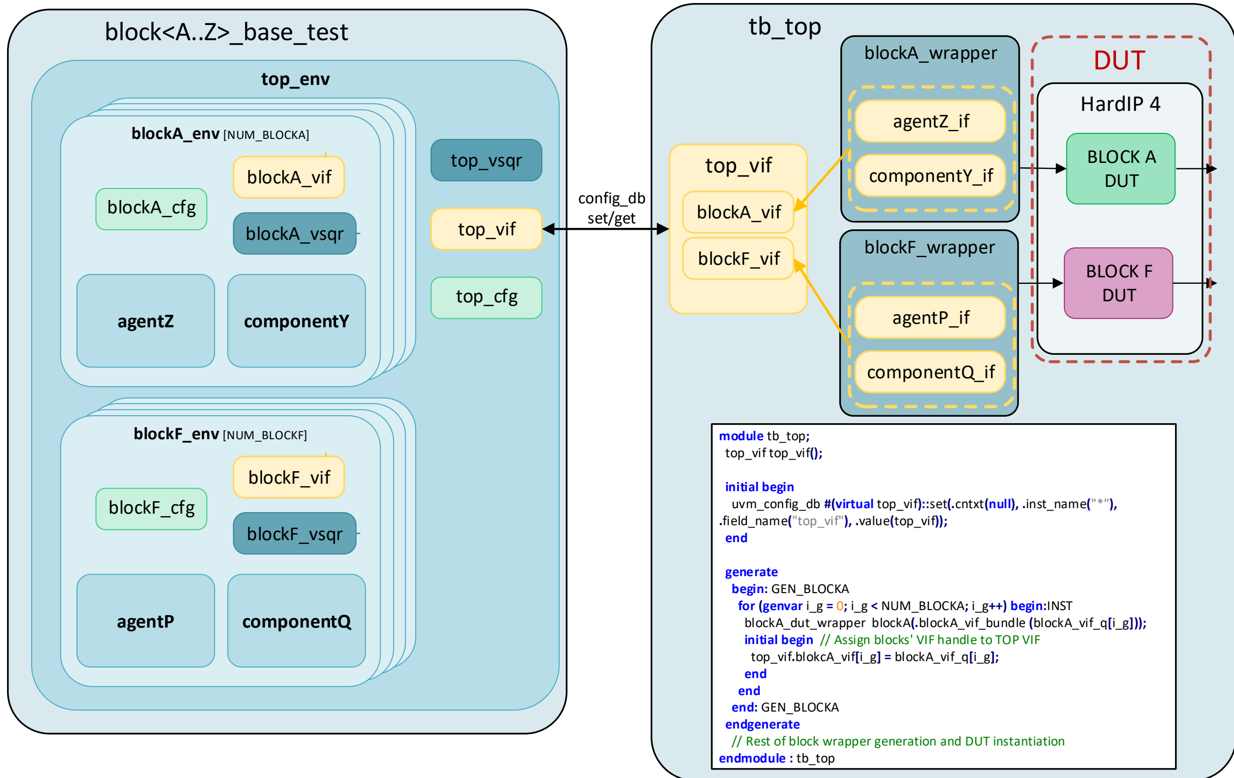

Figure 3 Testbench top integration example of one supported topology (HardIP 4) and testbench top sample code

3) Configuration

One of the main challenges that presents when providing coarse (pseudo-random testing) and fine (directed testing) configuration granularity is ensuring in both cases that it is consistent and coherent at all integration levels. One of the key benefits of this methodology approach can be seen especially at system-level, where we require certain inter-block dependencies to be aligned in both the blocks under test as well as in the verification components. This testbench architecture and methodology addresses:

1. Providing the capability for fine-grain configuration granularity while exploiting the benefits of pseudo-random verification (coarse-grain).
2. Ensuring the DUT configuration is valid and within the system requirements and expectations at all simulation levels including when inter-connecting multiple blocks together for system level simulations. This helps reduce TB debug time due to misconfiguration/time spent testing non-supported configurations.
3. Providing a maintainable and scalable configuration: modular code that facilitates horizontal and vertical reuse. Vertical integration is critical and often takes a significant amount of time in the verification schedule.

This methodology creates one main configuration *uvm_object* per environment and encapsulates any other block/subblock agent/component configuration object instance within the class centralizing all relevant information in a common place. In other words, the configuration object hierarchy mirrors the component hierarchy – see Figure 4. Similar to one of the recommendations from [2], we avoid using *uvm_config_db* with single fields and centralize all the *rand* variables into random configuration objects. It includes all required information related to testbench topology, test sequencing, DUT and stimulus configuration. This approach provides a modular and self-contained structure for ease of use and integration. It facilitates having access to all the relevant information by using object handles to navigate through the hierarchy while providing a coherent and consistent configuration view.

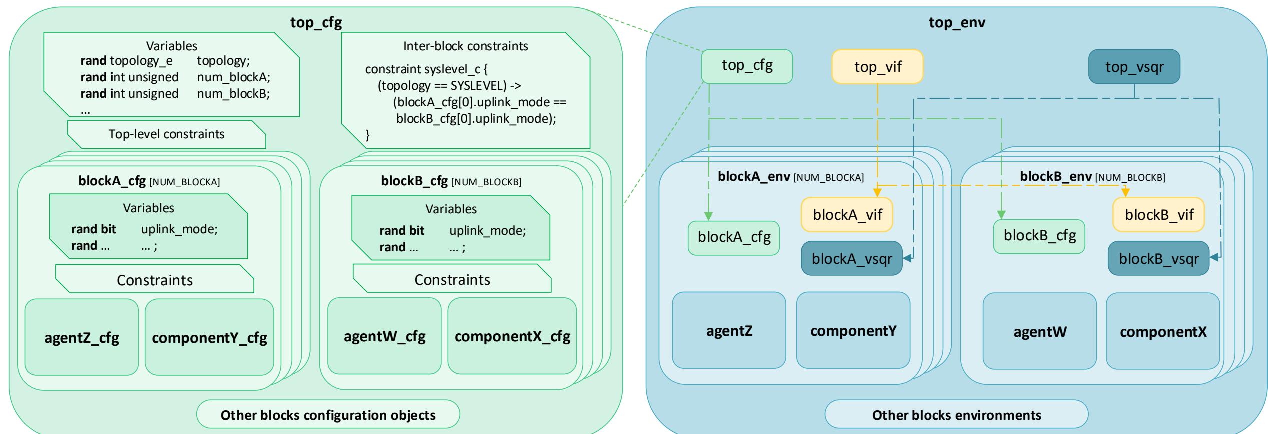

Figure 4 Configuration object and environment structure

As shown in Figure 5, the top-level configuration object defines as *rand* any other sub-component configuration object and creates them as part of the top-level configuration object *new* function. The top-level configuration object is being randomized at the start of the *build_phase* and by doing so, the entire configuration object hierarchy is randomized while satisfying the constraints at all different levels. This provides a coherent and valid view of the testbench topology, sequencing, DUT configuration and stimulus configuration to be used by the test. Randomizing and resolving inter/intra-block constraints as a whole is crucial when multiple blocks are

```
class top_cfg extends uvm_object;
  rand topology_e                   topology;

  rand blockA_cfg                   blockA_cfg_h[$];
  rand blockB_cfg                   blockB_cfg_h[$];
  //...
  `uvm_object_utils(top_cfg)

  function new(string name ="top_cfg");
    super.new(name);
    for(int unsigned i = 0; i < NUM_BLOCKA; i++)
      blockA_cfg_h[i] = blockA_cfg::type_id::create($sformatf("blockA_cfg_h%0d",i));
    for(int unsigned i = 0; i < NUM_BLOCKB; i++)
      blockB_cfg_h[i] = blockB_cfg::type_id::create($sformatf("blockB_cfg_h%0d",i));
    //...
  endfunction : new
  // Rest of class (functions, constraints, etc.)
endclass : top_cfg

class block_base_cfg extends uvm_object;
  rand int unsigned                     id;
  string                                dut_hdl_path;
  rand bit                              has_coverage;
  rand uvm_active_passive_enum          is_active;
  rand test_config                      test_cfg_h; // Defines main test sequencing used by the test
  rand stimulus_config                  stim_cfg_h; // Stimulus being used by the block if active

  `uvm_object_utils(block_base_cfg)
  // ...
endclass : block_base_cfg

class blockA_cfg extends block_base_cfg;
  rand agentA_cfg                   agentA_cfg_h;
  rand componentB_cfg               componentB_cfg_h;
  //Other BlockA component configs
  //BlockA DUT specific configuration knobs
  `uvm_object_utils(blockA_cfg)

  function new(string name ="blockA_cfg");
    super.new(name);
    agentA_cfg_h[i]     = agentA_cfg::type_id::create("agentA_cfg_h");
    componentB_cfg_h[i] = componentB_cfg::type_id::create("componentB_cfg_h");
    //Other blockA component configs creation
  endfunction : new
  // Rest of class (functions, constraints, etc.)
endclass : blockA_cfg
```

Figure 5 Top level and block level configuration object creation

inter-connected together to form a system and inter-dependencies need to be resolved. In this case, inter-block dependencies will be implemented in constraints as part of the top-level configuration object class (see Figure 4).

Because the configuration is randomized and resolved at the start of *build_phase* providing a valid solution from the overall possibilities space, this methodology expects that configuration to remain constant for the whole duration of the test hence not supporting later manual changes. Intervening at a later point partially modifying the configuration could potentially violate some of the constraints that were already resolved and cause unnecessary debug. This is an approach we do not recommend or use with this methodology. In order to intervene or provide input to the configuration for the randomization process, the main techniques used by this methodology are:

a) Using *pre_randomize* function and *plusargs*: run-time arguments are an efficient technique to feed in a specific fixed knob into the constraint randomization process without the need to recompile. If the run-time *plusarg* is provided, it will be processed as part of the *pre_randomize* function as we can see in Figure 6 example and *num_plusarg_c* will ensure that the random variable *num* takes the plusarg value. If on the other hand we are not making use of the *plusarg*, we expect the rand variable to be fully randomized within the allowed space defined by the configuration object constraints. In this case, we disable the *num_plusarg_c* constraint by setting its *constraint_mode* to 0.

```
class blockA_cfg extends block_base_cfg;
  rand int unsigned          num;
  int unsigned               num_plusarg;
  //...
  `uvm_object_utils(blockA_cfg)

  constraint num_range_c {
    num inside {[1:16]};
  }

  constraint num_plusarg_c {
    num == num_plusarg;
  }
  //...
  function void pre_randomize();
    super.pre_randomize();
    if ($value$plusargs("NUM=%d", num_plusarg))
      `uvm_info(get_name(), $sformatf("Reading plusarg NUM=%d", num_plusarg), UVM_MEDIUM)
    else // Disable num_plusarg_c constraint
      num_plusarg_c.constraint_mode(0);
  endfunction : pre_randomize
endclass : blockA_cfg
```

Figure 6 Example of configuration override using pre_randomize and plusargs

b) Extending the configuration object class and using UVM factory overrides: another option to provide input into the randomization process is by extending the configuration object class as we see in Figure 7.

```
class blockA_usecase0_cfg extends blockA_cfg;
  `uvm_object_utils(blockA_usecase0_cfg)

  function new (string name = "blockA_usecase0_cfg");
    super.new(name);
  endfunction : new

  constraint num_small_range_c {
    num inside {[1:4]};
  }
endclass : blockA_usecase0_test

class blockA_usecase0_test extends blockA_base_test;
  `uvm_component_utils(blockA_usecase0_test)

  function new (string name="blockA_usecase0_test", uvm_component parent);
    super.new(name, parent);
    // Constraining to set number within range 1-4
    set_type_override_by_type(blockA_cfg::get_type(), blockA_usecase0_cfg::get_type());
  endfunction : new

endclass : blockA_usecase0_test
```

Figure 7 Example of configuration override using factory override

In this case, the child class is applying a more restrictive rule to the variable *num* by a new constraint implementation in *num_small_range_c*. Then we apply a UVM factory override as part of the testcase

new function. The example in Figure 7 reduces the solution space in the extended class. Expanding the solution space can also be done by following the same technique and disabling the parent class contraints or overriding them with a new implementation. This methodology usually expects the block configuration object to capture the wider valid solution space to exploit the pseudo-random testing approach and it uses this technique to reduce the space trending towards a more directed testing approach.

4) Environment

Environments are dynamically generated to meet the overall DUT needs by creating only the required components and sub-components. The top-level environment holds dynamic structures of each block environment class that will be populated as required during the build phase, following the selected testbench configuration (Figure 8, A).

```
class top_env extends uvm_env;
  top_cfg                          top_cfg_h;

  blockA_env                       blockA_env_h[$];
  blockB_env                       blockB_env_h[$];
  blockC_env                       blockC_env_h[$];
  //...
  virtual interface top_vif        top_vif_h;

  `uvm_component_utils(top_env)

  extern function void build_phase(uvm_phase phase);
  extern function void connect_phase(uvm_phase phase);
  //Rest of functions
endclass : top_env

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

  // TOP-LEVEL ENV
  if(top_cfg_h == null) begin
    top_cfg_h = top_cfg::type_id::create($sformatf("top_cfg_h"),this);
    if (!top_cfg_h.randomize())
      `uvm_fatal(get_name(), "Failed to randomize top_cfg_h class" )
  end

  if(top_vif_h == null && !uvm_config_db#(virtual top_vif)::get(this, "", "top_vif", top_vif_h))
    `uvm_fatal(get_type_name(), "Interface top_vif_h not found")

  top_vsqr_h = top_vsqr::type_id::create("top_vsqr_h", this);

  // BLOCKS
   if(top_cfg_h.topology inside {TOP_HARDIP1, TOP_HARDIP2, BLOCKA_TOP}) begin
    for (int i=0; i<top_cfg_h.num_blockA; i++) begin
      blockA_env_h[i] = blockA_env::type_id::create($sformatf("blockA_env_h%0d",i), this);
      blockA_env_h[i].blockA_cfg_h = top_cfg_h.blockA_cfg_h[i];

      // Assigning VIF
      blockA_env_h[i].blockA_vif_h              = top_vif_h.blockA_vif[i];
      blockA_env_h[i].blockA_cfg_h.blockA_vif   = top_vif_h.blockA_vif[i];
    end
  end
  // Same as above for the rest of blocks
endfunction : build_phase

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

  top_vsqr_h.top_env_cfg_h  = top_env_cfg_h;
  top_vsqr_h.top_vif_h      = top_vif_h;

  //BLOCKA
  if(top_cfg_h.topology inside {TOP_HARDIP1, TOP_HARDIP2, BLOCKA_TOP}) begin
    for (int i=0; i<top_cfg_h.num_blockA; i++)
    begin
      // Assigning VSQR handles from block-level environments
      top_vsqr_h.blockA_vsqr_h[i] = blockA_env_h[i].blockA_vsqr_h;
    end
  // Same as above for the rest of blocks
endfunction : connect_phase
```

Figure 8 Top environment and sub-environment creation and handles setup

Consideration: a compile-time optimization is a more suitable solution for designs with high number of different block types. In this case, the environment declaration and creation is guarded by compile-time defines,

the environment uses static arrays and the associated environment package files are only compiled if the block is actually used by the selected topology. This approach greatly reduces the overall testbench compile time.

B. *Facilitating vertical and horizontal reuse and reducing testbench debug*

1) The OOP-style integration – no more *uvm_config_db* calls (* see footnote)

uvm_config_db get and set calls are based on strings that can use wildcards, which can be cumbersome and difficult to debug. As the database increases with more entries and because it is a string-based search, it creates a performance overhead that becomes unacceptable [2].

This methodology removes all (but two (*)) *uvm_config_db* calls often used to:

- Set configuration fields
- Pass configuration object handles
- Pass virtual interface handles
- Pass sequencer handles

The configuration object structure as per section A.3) automatically generates as part of one single phase all configuration objects providing a consistent view with all the information required to build the testbench components, configure them, the DUT, stimulus and test flow. It is the environment which takes a key role in this process. The environment as part of the build phase ensures that any component created will have:

- The configuration objects required by passing down the handles to all the different components and agents (Figure 8, B)
- The interfaces they need. If this environment is the top-level one, it retrieves a virtual bundle interface by a single *uvm_config_db* call (*) and assigns the relevant interface handles down the hierarchy as required. If it is not the top-level environment, it will be provided with the virtual interface bundle at this level and distribute the sub-components interface handles following the same strategy as before (Figure 8, C)
- The sequencer. Following a similar self-contained structure as the configuration objects and virtual interfaces, the environment virtual sequencer class contains handles to all the sub-component sequencers (Figure 9). Once created, the environment ensures that all components created within have their sequencers set accordingly during build phase (Figure 8, D).

```
class top_vsqr extends uvm_sequencer;
  top_cfg                               top_cfg_h;
  virtual interface top_vif             top_vif_h;

  blockA_vsqr                           blockA_vsqr_h[$];
  blockB_vsqr                           blockB_vsqr_h[$];
  //...
  `uvm_component_utils(top_vsqr)

  // Class functions
endclass : top_vsqr
```

Figure 9 Top level sequencer class

- Sequences retrieve all necessary handles from the parent sequencer before they start (Figure 10).

```
function void blockA_vseq::set_handles();
  blockA_cfg_h       = p_sequencer.blockA_cfg_h;
  blockA_vif_h       = p_sequencer.blockA_vif_h;
  test_cfg_h         = blockA_cfg_h.test_cfg_h;
  //Other handle retrievals
endfunction : set_handles
```

Figure 10 Handles setup in sequences

2) Constrained-random configuration

Ensuring that the configuration is aligned across multiple blocks through constraints reduces testbench debug time due to DUT misconfiguration/unsupported system setup. Any configuration inconsistency will make the simulation fail during randomization i.e. at the start of the test simulation, saving time by failing early.

(*) Note that a single *uvm_config_db* set/get pair was required to pass the virtual interface bundle from the testbench to the environment

3) Base testbench classes automatically generated

Because of the systematic approach of this methodology, we were able to automatically generate all base classes and testbench skeleton at the start of the project securing a strong foundation with which to begin. Leveraging from OOP inheritance, we were also able to unify all block base classes maximizing reuse and maintainability.

4) SVA and the "no absolute hierarchical probes" strategy

The approach taken to SystemVerilog Assertions (SVA) is to use *binds* where possible and relative hierarchical probes as part of the checking modules. This helps with integration and reuse by providing an adaptable method to connect to the DUT and also removing fixed dependencies created by absolute probes. Use of any absolute hierarchical probes should be avoided to facilitate both vertical and horizontal integration.

C. Maximize reuse of test sequences at the different integration levels

1) Sequences

A base sequence class (**block_base_vseq** in Figure 12) was used as a template for all the blocks to implement and unify the different stages of the test sequence (power-on-reset, enable/control settings, register configuration, applying stimulus/sending transactions, wait for output transactions, etc.). This systematic structure allowed us to reuse unit-level sequences for system-level integration.

2) Tests

A common approach is to have components/environment setup code as part of the test classes. Following that approach makes the testbench difficult to maintain, reuse and prone to errors as this code is typically not run at the higher integration levels. This methodology implements environment components to be self-contained keeping tests simple and mainly targeting configuration overrides to constrain the configuration to suit the test (example in Figure 7).

As part of the main base test class (**top_base_test**), we instantiate the top-level environment. Because of the OOP approach and systematic integration structure, we have access to any child class without incurring any significant testbench performance penalty.

a) Unit-level

Each block base test extends the main base test class and creates handle aliases to their environment, configuration object and sequencer for ease of use as shown in Figure 11. A *plusarg* ID option is used to select which specific block instance we run the test sequence in, which defaults to instance 0 for unit-level as there is only a single block DUT instance.

```
class blockA_base_test extends top_base_test;
  blockA_env               blockA_env_h;
  blockA_cfg               blockA_cfg_h;
  blockA_vsqr              blockA_vsqr_h;
  blockA_base_vseq         blockA_vseq_h;

  `uvm_component_utils(blockA_base_test)
  //...
endclass : blockA_base_test

function blockA_base_test::new(string name, uvm_component parent);
  super.new(name, parent);
  top_vseq_h = top_base_vseq::type_id::create("top_vseq_h");
  get_env_id_plusargs();
endfunction : new

function void blockA_base_test::connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  // "Alias" handles for unit level testing
  blockA_env_h  = top_env_h.blockA_env_h[id];
  blockA_vsqr_h = top_env_h.top_vsqr_h.blockA_vsqr_h[id];
  blockA_cfg_h  = top_cfg_h.blockA_cfg_h[id];
endfunction : connect_phase

function void blockA_base_test::get_env_id_plusargs();
  if ($value$plusargs("BLOCKA_ID=%d", id))
    if ( (id < 0) || (id >= NUM_BLOCKA))
      `uvm_fatal(get_name(), $sformatf("Specified BLOCKA_ID is outside bounds. BLOCKA_ID: %d. Range 
permitted: [%d,%d]", id, 0, NUM_BLOCKA-1))
    else
      `uvm_info(get_name(), $sformatf("Overriden by plusarg BLOCKA_ID = %0d",id), UVM_MEDIUM)
endfunction : get_env_id_plusargs
```

Figure 11 Block unit-level base test class with environment ID plusarg

b) Top-level

In the case of top-level verification, this *plusarg* ID will be specified by the regression list to target all the various instance IDs that form the HardIP. By following this approach, we achieved a 100% vertical testcase reuse, from unit-level to top-level, as well as reusing horizontally across all possible block instances in the HardIP. Note that 100% vertical reuse may not be possible for all architectures, but this approach should maximize reuse and ease of integration at the next level.

c) System-level

The system-level approach is slightly different as there are inter-dependencies in the configuration of the blocks forming the system as well as the sequencing. The main block sequences were also fully reused to form the final system level sequence as shown in **syslevel_vseq** in Figure 12. The system-level sequence, following the same virtual sequence tasks as the base class, reuses blockA and blockB implementation from the unit-level virtual sequence implementation. New constraints were also implemented as part of the top-level configuration object to ensure alignment in configuration across inter-connected blocks (e.g. Figure 4) and allowing us to set all the component configurations accordingly (*UVM_ACTIVE* / *UVM_PASSIVE*).

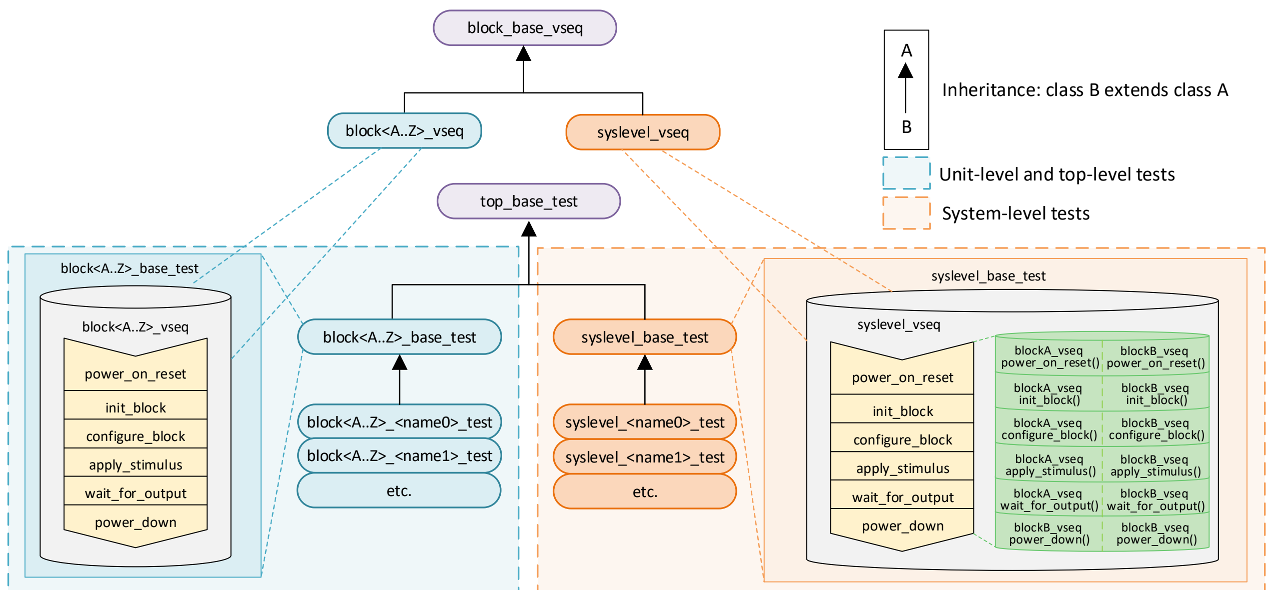

Figure 12 Virtual sequences and test inheritance structure and test sequencing implementation

III. Conclusions

This testbench architecture and coding methodology was implemented to successfully verify Xilinx RFSoC DFE [1] and it follows generic guidelines that can be applied to other type of ASIC and SoC designs. Its adoption achieves a single UVM testbench with maintainable and reusable code to support all levels of integration while enabling parallel verification execution and ease of integration. By using the proposed methodology, we achieved 100% vertical and horizontal test reuse from unit-level to top-level. Its flexibility and configurability de-risks potential top-level DUT changes as well as providing a flexible framework for new custom system-level use cases.

References

[1] Xilinx RFSoC DFE: https://www.xilinx.com/products/silicon-devices/soc/rfsoc/zynq-ultrascale-plus-rfsoc-dfe.html

[2] R. Edelman, C. Spear, “UVM – Stop Hitting Your Brother Coding Guidelines”, DVCON US 2020

[3] IEEE 1800-2017 – IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language