Configurable Testbench (TB) for Configurable Design IP

Kilaru VamsiKrishna
Sushrut B Veerapur
Cadence Design Systems

Abstract- Design IPs are highly configurable and offer multiple architectural flavours to suit various custom requirements. To support various architecture flavours of Design IP, IPs are built hierarchically by integrating multiple reusable sub-blocks, which gives the requirement and scope for verification at multiple hierarchies. Configurable TB should support various modes of IP and vertically scalable. TB hierarchy should be like RTL hierarchy and all TB components should scale for the same.

This paper demonstrates the methodology to build a *Highly Reusable and Highly Scalable Testbench (TB).*

Keywords - UVM, Configurable Testbench, Scalable Testbench

I. INTRODUCTION

A typical IP provides multiple configuration options. Design IPs are configurable with respect to protocol features and the end usage of IP. Design IP offers more architectural flavours and offers thousands of configurable features. Registers in an IP can be programmed through an APB or AXI or JTAG Interface. AL (Client or System) interface can be any of AMBA-CXS/AMBA AXI/ AMBA-AHB or any other custom interface and the sequences which drives the data at various interfaces should be reusable at all the client interfaces. A Two layered sequence approach is used to achieve this.

As design IP has configurable system interfaces and traffic generation at each system/client interface is different. There is a need to architect the sequences for reusability at various system interfaces independent of the nature of the interface.

Bases on the IP end usage, IP can be integrated and configured to run at various hierarchies like controller only mode or controller with PHY or in a test chip. Testbench must be scaled to run at all the sub system to system level and at the same needs a requirement of a simple integrated testbench.

To support configurable design IP, TB must be built hierarchically. Single Testbench configurable for all Metric Driven Verification and act as an Integration Testbench would be an ideal solution. If the Sub IP supports various modes of the TB, TB env must be instantiated based on the supported modes. To avoid TB dependencies depends on the mode of the Design IP, and to support multiple features necessitates the below TB requirements.

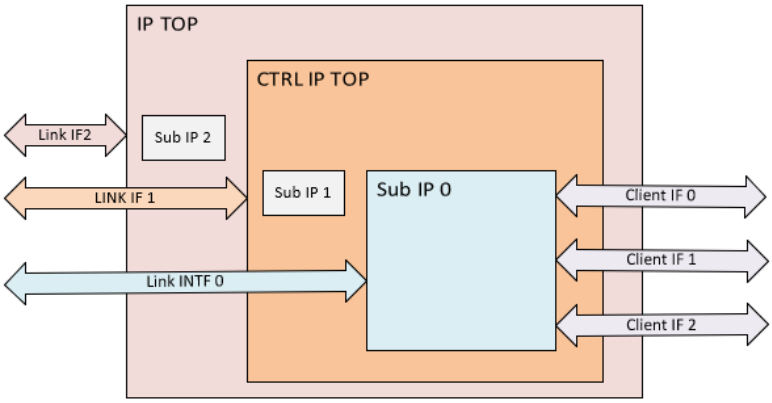
- High configurability at the lowest feature level
- Constraints Solver for Sequence Items and Configuration Classes
- Layered Testbench
- Design parameter configuration control at compile time
- Coverage configuration control for meaningful coverage avoiding full open design scope
- Coverage merge conflicts across configurations

Advanced OOPs concepts, Policy classes, Typedef, adapter and Proxy class principles are essential to build a configurable testbench.

II. TESTBENCH CONSIDERATIONS

The Design IP can be configured through Parameters and Straps during the integration independent of protocol feature configuration. The design architecture specification changes based on this inputs and parameters. To qualify the IP supporting various architectural flavours, randomization of parameters is essential.

- Example of a PCIe and CXL controller configurations
  - Based on System Bus Architecture
    - AXI, CXS
  - Register Configuration
    - AXI Lite, APB
  - Based on Protocol Link Type
    - PIPE4.x, PIPE5.x, PIPE6, Serial
  - Based on Protocol
    - PCIe, CXL

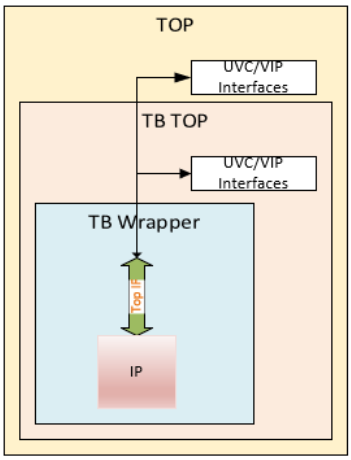


A. Parameter Randomization

To Verify all the legal and valid combinations of parameterized IP (Predefined combinations) need to randomize the legal space of the parameters. Encapsulate all parameters of the Design IP as a rand members of SV object and set the values through constraints. Create a function which renders the Parameter package from the randomized class members. Split the simulation flow into two steps and in 1st step generate the parameters package and use the generated package to override parameters of Design IP in the 2nd step simulation flow.

B. Testbench and IP Integration with interfaces

Based on the hierarchy of the IP, RTL ports can change. Maintaining the ports change is not trivial if the differences are quantifiable. Use a TB wrapper generation script to create a common top interface, TB module wrapper and connect the IP and interface. Top Interface connects to Sub IP0 or CTRL IP TOP or IP TOP based on the IP configuration through script. Based on the IP configuration, connect the UVC/Sub IP Interfaces to top interface and this allows TB wrapper agnostic of port changes. Layered TB TOP ensures the interface connectivity of RTL with VIP/UVCs. Ensure ENV paths and HDL hierarchy paths are configurable to set the interfaces to UVCs.



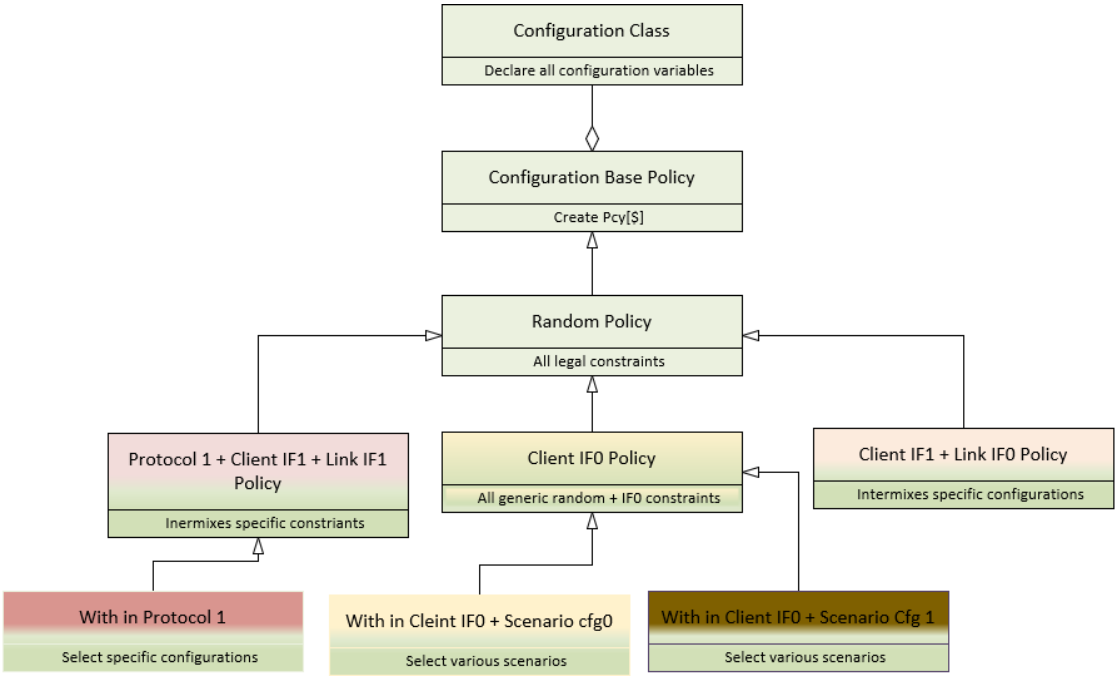
C. Configuration class Considerations

Design IP can have multiple configuration objects based on the protocol and TB requirements and constraints every configuration object need to change based on the IP configuration. Some constraints can be mutually exclusive, and the weights of the rand variables require a change based on the design architecture requirements. Maintaining all the constraints in the same class in a flat hierarchy may not be suitable to scale the Testbench.

Scenario configuration-controlled Policy class strategy employed for clean configuration segregation. Scenario configuration is a configuration object which randomly selects the multiple legal scenarios and control the complete testbench. To fine control the variables in scenario class command line arguments are used. Command line arguments processor forms an args package. Make args creates defines and add the controls to Command line args which ensure always Design IP and Testbench are aligned.

Policies are created dynamically and pushed to the associated configuration class. Multiple policies can be applied to get the fine control on the configuration. The below is an example of the policy class tree.

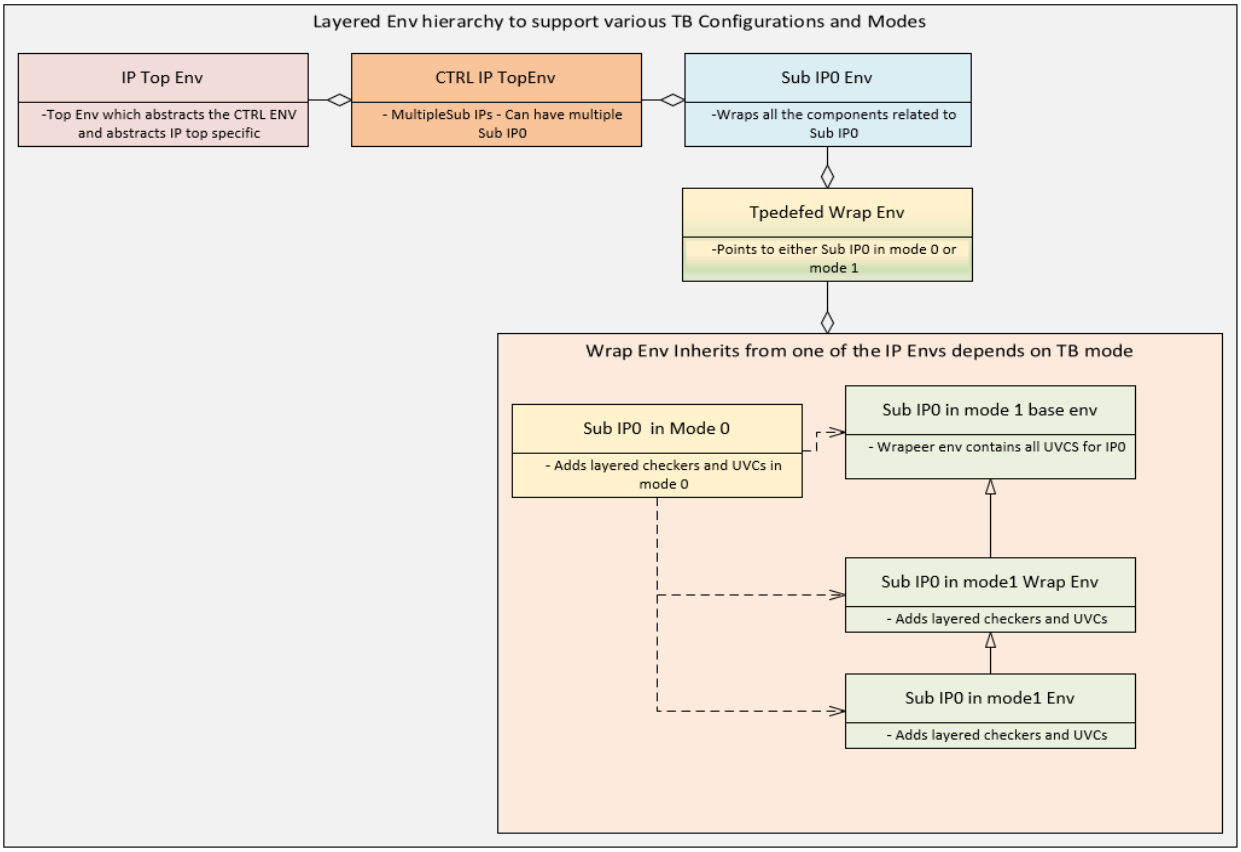
```
class cdns_uvm_base_rand_policy #(type T = uvm_object) extends uvm_object;
  //------------------------------------------------------------------------------
  // Field: item
  // Handler for object which will be randomized using constraints defined
  // in this sub-classes of this one. Must be setup before randomization.
  //------------------------------------------------------------------------------
  protected T item;

  //------------------------------------------------------------------------------
  // Constructor: new
  // This constructor set default name of object.
  //------------------------------------------------------------------------------
  function new(string name = "cdns_uvm_base_rand_policy");
    super.new(name);
  endfunction : new

  //------------------------------------------------------------------------------
  // Function: set_item
  // Setter for item handler.
  //------------------------------------------------------------------------------
  virtual function void set_item(T item);
    this.item = item;
  endfunction : set_item

  //------------------------------------------------------------------------------
  // Function: get_item
  // Returns handler to item.
  //------------------------------------------------------------------------------
  function T get_item();
    return item;
  endfunction : get_item

  //------------------------------------------------------------------------------
  // Function: pre_randomize
  // Checks whether item handler has been setup.
  //------------------------------------------------------------------------------
  function void pre_randomize();
    if (item == null)
      `uvm_error(get_type_name(), $sformatf("Handler 'item' is not set"))
  endfunction : pre_randomize
endclass : cdns_uvm_base_rand_policy

//------------------------------------------------------------------------------
// Method: push_policy
// Pushes back handler to policy object given in arguments to policy queue.
//------------------------------------------------------------------------------
function void push_policy(cdn_pcie_strap_base_policy policy);
  policy.set_item(this);
  pcy_q.push_back(policy);
endfunction : push_policy
class cdn_pcie_strap_random_policy extends cdn_pcie_strap_base_policy;

    constraint k_datapath_wd_c { item.k_datapath_wd inside {4, 8, 16}};
    constraint k_alt_prot_neg_support_c { item.k_alt_prot_neg_support == 0;}
```



D. Testbench Hierarchy and Layered Testbench considerations

Testbench hierarchy must be maintained to isolate multiple layers depends on the Design IP configuration. As shown in the figure 1, the Design IP can have a multiple hierarchy and testbench maintains the similar topology. The primary env, builds all the essential UVCs, components required for IP0(lowest possible RTL Top). All other ENV classes inherited from one another to maintain the topology like Design IP configuration. Typedef is used as wrapper class to instantiate the applicable class based on the selected configuration. Each env class has its own configuration and maintain the hierarchies. ENVs and containing agents are configured in active/passive based on configuration.

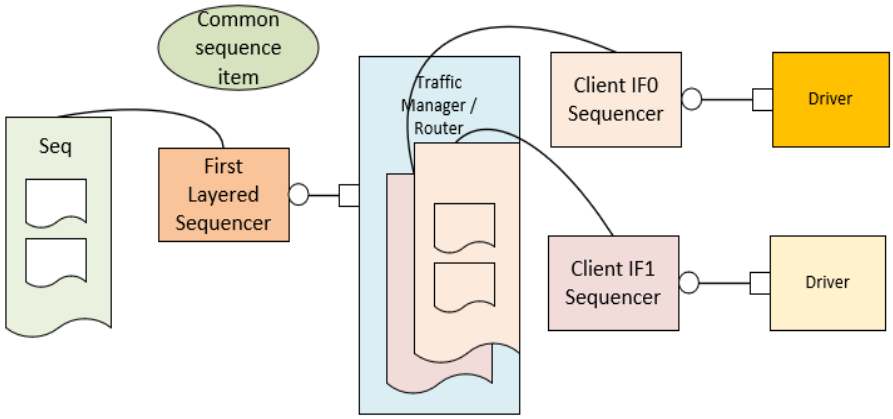
```
// In the package
`ifdef IP_TOP
   typedef  cdn_ip_top_env       cdn_dev_env;
`elsif CTRL_IP_TOP
   typedef  cdn_ctrl_ip_top_env  cdn_dev_env;
`elsif SUB_IP_TOP
   typedef  cdn_sub_ip_top_env   cdn_dev_env;
`endif

// up the hierarchy
class cdn_top_env extends uvm_env;
  cdn_dev_env m_dev_env;
  //...........
endclass
```



D. Layered Traffic Sequence Considerations

DIP Can support multiple BUS architectures and traffic patterns changes based on the interface protocol. Traffic sequences at various client interfaces must be common and should be independent of design architecture. To achieve this, create a common sequence item and layer the sequences to convert to bus specific sequence items. A router or traffic manager is used to route the sequences based on selected configurations.

- Other sequence considerations
  - Register programming sequence is common and traditional register flow support this. Register models are build based on the Design IP configuration and Adapters, monitors, sequencers are pointed accordingly.
  - Sub sequences are controlled in virtual sequence depends on the configuration and provide hooks to start various architecture specific sequences.
  - Use typedef to forward reference sub sequences in the main virtual sequence



E. Scoreboard Considerations

Like hierarchy of IP, scoreboards are added hierarchically from one level to other level. All the checkers, module monitors and sub env's scale based on configuration. Interfaces are bind to RTL hierarchically using uvm harness technique. All the sub module level assertions naturally scale to next level.

F. Coverage Considerations

In the metric driven verification functional coverage of an IP place a critical role and scaling the coverage based on the design IP configuration is challenging. To avoid the coverage merge conflict, coverage control config object is passed during the coverage construction. The proxy/interface class holds the maximum possible design configurations unlike the dynamic configuration which changes per seed during the regression. With the proxy class coverage will scale like design IP configurations.

```
covergroup cg (cdn_cfg_proxy max_cfg_proxy) with function sample(cdn_cfg i_cfg);

    cp_value : coverpoint value iff(i_cfg.enable) {
    bins values_supported[] = {[0: max_cfg_proxy.max_values]};
  }
endgroup : cg

// In the coverage collector
cg cg_h;
cg_h = new(max_cfg_proxy);
```

III. APPLICATIONS

This framework can be used for any complex IP verification. This methodology considers the requirement of end to end testbench development considering RTL Integration, Testbench Topology, Configuration classes, Sequences, Checkers and Coverage.

IV. SUMMARY

The concepts used in developing the *Configurable Testbench* are simple and effective.

- OOPS
- Design Patterns (Policy, Proxy/Interface, Adapter/Bridge)
- Typedef Classes
- Common Sequence Item
- Layering Concepts (Env, Sequences)
- Traffic Managers
- Testbench Hierarchy Partitioning

The mentioned techniques consider most requirements of industry IP testbenches, which highlights how structurally well-architected and well-implemented single testbench can scale for multiple verification requirements at various hierarchy and levels of the design.

REFERENCES

[1] I Didn’t Know Constraints Could Do That, John Dickol, DVClub Europe
[2] https://verificationacademy.com/forums/ovm/randomizing-module-parameters