Statically Dynamic or Dynamically Static? Exploring the power of classes and enumerations in SystemVerilog Assertions for reusability and scalability

Sachin Scaria
sachin.scaria@intel.com
Intel Technology India Pvt. Ltd.
SRR Elite, Sector 3, Bellandur, Bengaluru,
Karnataka, India 560103

Sreenu Yerabolu
sreenu.yerabolu@intel.com
Intel Technology India Pvt. Ltd.
SRR Elite, Sector 3, Bellandur, Bengaluru,
Karnataka, India 560103

Don Mills(Presenter only)
don.mills@microchip.com

***Abstract*-This paper discusses about the usage of class data type to assist assertions coding for SoC level verification, making the code modular and reusable by type override feature in SystemVerilog. This implementation can be further extended to applications where complex data structures are to be used with low performance impact on the simulators.**

Keywords—assertions; sva; class; SystemVerilog; type override; module

I. INTRODUCTION

"There's a way to do it better—find it." — attributed to Thomas Edison

There is always a question of meeting the two ends in static and dynamic together in SystemVerilog. Even though we have many features in LRM for SystemVerilog, which address the meeting point of static and dynamic components such as virtual interfaces, assertions have been left unattended, to an extent. How better can we make something dynamic, (oh, it's static too!!) making the code modular and portable, and thereby lowering the chances of committing a human error.

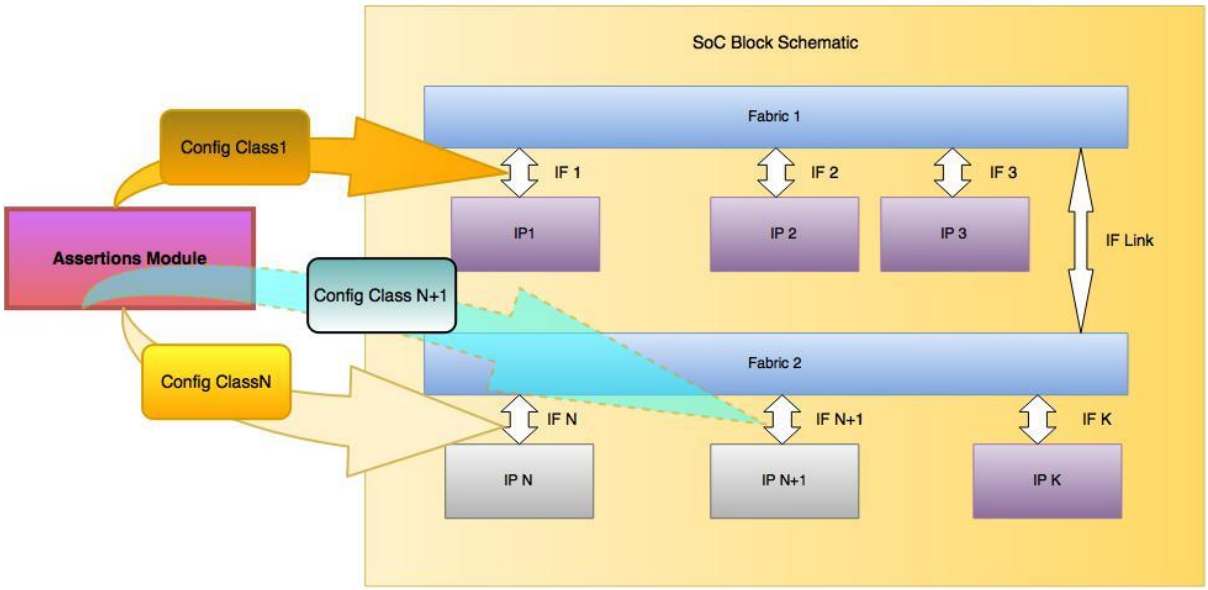
When there is a handcrafting of assertions, which are expected to be reused across projects and revisions, the question of modularity and portability gains more significance. But this prominence did not give any boos into the assertion syntax and we did not have a way of writing an assertion inside a SystemVerilog Class until LRM 2012.

This paper will discuss about developing a generic assertion for a particular identified scenario, without changing the template of the assertions. This means, whenever the assertions have to be plugged into a particular hierarchy in the design, there is no change in the assertion module, but the variables inside the assertions, will get modified by passing a data type encapsulation via classes.

This approach will call for maintaining only a single file, which has the set of assertions, with other required/specified changes being done at the time of instantiation. In a SoC environment, where numerous IPs can get added, with changing product/revision/design rules, the number of changes/replications required for assertion

files can get amplified, making way for human errors. The approach described in this paper can help in reducing these human errors given a need to add similar interfaces/IPs into the SoC design and verifying the DUT's correctness. In this approach, when specification changes, one needs to only update a few fields in the class encapsulation and the assertion is all set to plug-in and flush out design bugs in the given SoC environment. Even if the assertion has to be modified, the modification is required only in a single module file, which eventually applies to all the Testbench/Environment interfaces through SystemVerilog bind construct.

Below diagram depicts the case, where this approach has been used:



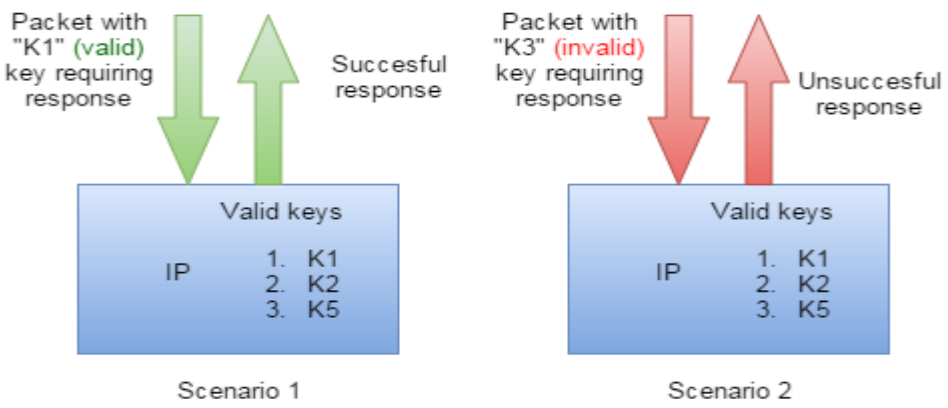
The use case was a critical feature that got added to multiple blocks in a SoC, that will continue to exist for a few generations to come, of course with specification changes. Here all the interfaces works on the same set of protocol, but will have differences on the packet acceptance/response criteria. Making each block aware of the configuration is tedious without thinking out of the box. Since normal usage would need one file specific to each interface, or at least "include" files would be necessary, which in turn necessitates more than one assertion module.

In the proposed approach, this challenge is surpassed using type override in module using SystemVerilog class (config class in figure above), and passing enumerations inside virtual classes to further encapsulate the variable types. With this we can achieve complex data structures and the advantage is that the value/type itself can be changed without touching the actual assertion module.

II. IMPLEMENTATION IN DETAIL

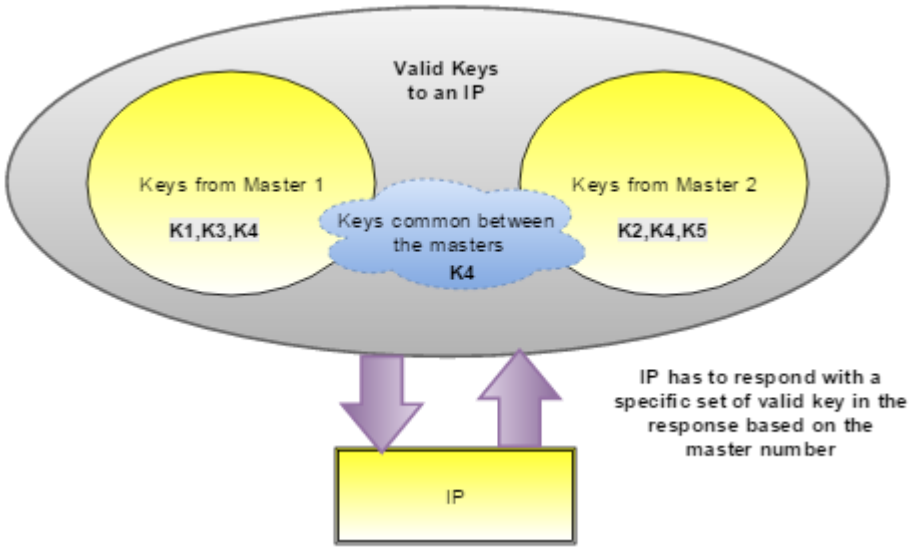
The requirement from the development had started as given below,

"*All IPs should respond to an incoming transaction of a type that needs a response, provided that the key value inside the packet received matches one of the configured values hardwired in the IP. Otherwise, it should respond with a packet with an unsuccessful response.*"



The image shows the valid keys in possession of a particular IP, and the type of response it evokes, for an incoming packet, based on the validity of the key in the packet. In a similar manner, different IPs will respond differently based on the set of allowed keys present inside the IP.

Describing the bigger picture of what is depicted above, here K1, K3, K4 are valid for master cluster 1 and K2, K4, and K5 are valid for master cluster 2 for the particular IP under consideration. When K4 is received at the IP, **it will be able to understand the source master cluster which initiated a packet with K4, from the protocol signals and not based on the Key alone**.



Another scenario below describes details on how the master keys are made visible to the IP. The IP will be able to understand the source master from the protocol signals and not based on the Keys alone.

Pseudo code can be found below,

*/*Valid Keys for group1 initiator. These classes can be declared anywhere outside the assertion block. Classes are required here to encapsulate the enumerations having same value (4 in the scenario described). SystemVerilog doesn't allow the same enumeration value to be used twice in the same compile scope. Classes comes to rescue in creating the enumerations in two different compile scopes.*

/

```
virtual class grp1;
 typedef enum logic [7:0 ] {
   K1  = 8'h1,
   K2  = 8'h3,
   K5  = 8'h4  // 8'h4 is  in group2 as well
 } valid_keys;
endclass
```

```
//Valid Keys for group2 initiator
virtual class grp2;
 typedef enum logic [7:0 ] {
   K1  = 8'h2,
   K3  = 8'h4,  // 8'h4 is  in group1 as well
   K5  = 8'h5
 } valid_keys;
endclass
```

```
module IP_check #( type T1 = int, type T2 = int )(my_vif vif) ;
…
```

```
//declaring variables for the enumeration definition. This gets overridden at the time of instantiation
typedef T1::valid1_enums   my_valid1_enum_def;
my_valid1_enum_def         my_valid1_enum;

typedef T2::valid2_enums   my_valid2_enum_def;
my_valid2_enum_def         my_valid2_enum;

…
// rest of the assertion code
//Usage of .name () inbuilt function in enumeration was used for
//  finding the validity of incoming key and outgoing key

//assuming vif.key is a bus declared in the vif interface
logic  valid1_key;
assign my_valid1_enum =  my_valid1_enum_def'(vif.key)
assign valid1_key     = (my_valid1_enum.name () != "");

//assuming vif.key is a bus declared in the vif interface
logic  valid2_key;
assign my_valid2_enum =  my_valid2_enum_def'(vif.key)
assign valid2_key     = (my_valid2_enum.name () != "");

//sample sequence and assertion code given as two groups itself for better readability
sequence check_key_grp1 ;
      @(posedge vif.clk)
      ((vif.cmd_start) && (vif.grp == 2'b0) && (valid1_key));
endsequence
```

```
   sequence check_key_grp2 ;
         @(posedge vif.clk)
         ((vif.cmd_start) && (vif.grp == 2'b1) && (valid2_key));
   endsequence

   // assertion for grp1
  property check_grp1_cmpl;
     logic [7:0] local_tag;
     @(posedge vif.clk)
     disable iff (!vif.rst_b)
     (check_key_grp1.triggered, local_tag = vif.tag) |->
       (vif.opcode == 2'b0) && (vif.resp_tag == local_tag);

          //opcode    2'b0 -> successful operation
      endproperty

   // assertion for grp2
   property check_grp2_cmpl;
     logic [7:0] local_tag;
     @(posedge vif.clk)
     disable iff (!vif.rst_b)
     (check_key_grp2.triggered, local_tag = vif.tag) |->
       (vif.opcode == 2'b0) && (vif.resp_tag == local_tag);
          //opcode    2'b0 ->  successful operation
  endproperty

   …
endmodule
```

/ Below piece of code is the heart of the implementation, where the classes declared outside of the assertion module is passed into each instance according to the specification, and the same is internally used for the assertion check. This helps in maintaining only one assertion file and the enumerations can be overridden at the time of instantiation. */*

```
   module tb ();
     IP_check #(.T1(grp1),.T2(grp2)) IP1(vif1); // override using the encapsulated class itself
     IP_check #(.T1(grp3),.T2(grp4)) IP2(vif2);
   ...
   endmodule
```

III. Results

This is an experimental work and the feature described is still not implemented completely in all simulators. But enhancements are already filed and the proof of concept is already working with one of the Big 3 simulators. The full working code sample is captured in appendix

IV. References

[1] Sreenu Yerabolu, Sachin Scaria, , “Thank you VGEN - Complex SoC Assertion Qualification Turnaround Time from Hours to Minutes” SNUG Silicon Valley, 2016.

[2] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," Section 16.8, 16.12, Assertions.

[3] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," Section 19.8.1, Sample function override.

[4] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," Section 23.10.3, page 700, example about type override using class.

APENDIX

Capturing a fully functional code describing the concept mentioned in the paper. This can be compiled and simulated as is

```
virtual class a;
  typedef enum logic [7:0 ] {
    K1 = 8'h01,
    K2 = 8'h08,
    K3 = 8'h09
  } valid_enums;
endclass

virtual class b;
  typedef enum logic [7:0 ] {
    K2 = 8'h9
  } valid_enums;      // enumerations are having the same name here in class a and b
endclass

// Here both class a and b are having a common enumeration inside valid_enums (K2).
// This can only be compiled when the enumeration is encapsulated/abstracted using a
//compile scope as per SV LRM.
//
// Reason for using two classes for declaring the enumerations is only to encapsulate the enumerations.
// virtual keyword is used here, for restricting anyone to use these classes inside the code accidentally.
// Currently the members are accessed using "::" scope resolution operator

module enum_test ();
  parameter type T = b;
  logic [7:0] local_enum;
  string my_string;

// below two lines of code helps in declaring the type and the usage.
//The elaborated code will have the member "T" getting overridden during instantiation
//(during parameter override)
  typedef T::valid_enums   my_valid_enum_def;
  my_valid_enum_def   my_valid_enum;

  initial begin
    local_enum = 8'h9;
    my_valid_enum = my_valid_enum_def'(local_enum);
    my_string = my_valid_enum.name() ;
    $display ("value is %s",my_string);                    // Prints K3 since the class is overridden to "a"
    local_enum = 8'h9;
    $display ("value is %0d", b::valid_enums'(local_enum)); // Prints  K2 since class b is used directly
  end

endmodule

module tb_enum_test ();
 enum_test #(.T(a)) dut_enum_test () ;      // "T" get overridden to "a"
endmodule
```