Exquisite modeling of verification IP: Challenges and Recommendations

Anuradha Tambad (Anuradha.tambad@lsi.com),
Subashini Rajan (Subashini.rajan@lsi.com),
Shivani Upasani (Shivani.upasani@lsi.com),
Prashanth Srinivasa (Prashanth.srinivasa@lsi.com),
Imran Ali (Imran.ali@lsi.com)
LSI India R&D Pvt Ltd, Bangalore, India

Adiel Khan (adiel@synopsys.com)
Synopsys (Northern Europe) Ltd, UK

Abstract— Standard verification methodologies like UVM/VMM provide guidelines for developing reusable verification IPs. But for complex designs with huge amount of data transfers and whose interface protocol could be upgraded for better performance, special care needs to be taken for architecting verification IP's. Further, large number of interdependent parameters in design poses another challenge of complex stimulus generation to the constraint solver and to the random verification. This paper describes the challenges faced and the innovative steps followed in developing an efficient reusable VIP. These steps enable us to reduce the overhead on tool constraint solver engine, to save system run-time memory and to reduce the rework when interface protocol changes.

Acronyms:

SoC: System On Chip
VMM: Verification Methodology Manual
UVM: Universal Verification Methodology
DMA: Direct Memory Access
DUT: Design Under Test
VIP: Verification Intellectual Property

I. INTRODUCTION

With the growing size of complex SoCs, the VIPs involved in the verification have to be more efficient, faster and reusable to traverse various design features at different hierarchies. The most important feature of a VIP is to have an efficient generation mechanism to randomize and control the stimulus through right set of constraints.

When there is huge amount of data transfers involved, it becomes equally important to develop efficient data models along with other components considering the system memory consumption. In addition, constantly evolving specification during design cycle, require repeated updates to the verification IP. Any reusable VIP should be built to accommodate the changes with minimal effort and concentrate more on the verification aspects, thus reducing the verification effort and time to market

Following are the challenges encountered while developing the reusable verification IP.

1. Simulator takes longer time to solve large set of inter-related constraints, since all the constraints are solved in parallel. If the constraints are replaced with procedural code, maintaining interdependency between the parameters and controlling the transaction from test case becomes very difficult.

2. Generating large amount of pre-generated data for each transaction consumes significant runtime memory since each transaction is stored in reference models till the scoreboard check is complete. Hence the memory requirement can easily go beyond the available system runtime memory.

3. Frequent changes in interface protocol for architectural reasons (better performance etc) affects the verification IP development as well. A high degree of configurability has to be provided to the verification IP to enable reuse at sub-chip or chip levels and in different modes.

The following sections describe how all these challenges were addressed in the VIP without compromising its quality and performance.

II. CHALLENGES FACED AND RECOMMENDATIONS

A. Efficient way to break down complex constraints in different randomization phases:

Constrained random verification environment needs to take charge of the inadequacy of the constraint solver in

solving of large number of inter-related parameters in parallel. When the number and size of properties to be constrained keeps increasing, the constraint solver times out as it takes longer to arrive at a possible value for the random properties. Replacing the parallel constraint code with procedural code increases the lines of coding and is also prone to errors mainly because of interdependent constraints. On randomization of the class, independent member properties and inter-dependent properties are solved in parallel. In order to lessen the burden on the constraint solver, the different type of class properties like independent, inter-dependent and dependent fields are segregated for procedural and parallel generation.

In Figure.1 Constraints '*size_c*' and '*cfg_c*' indicate '*cfg_descr*' and '*size_decoder*' are inter-dependent.

```
rand bit[5:0] size_decoder;

rand bit [3:0] cfg_descr;

  constraint size_c { (size_decoder == 3) -> (cfg_descr
== 1); };
 constraint cfg_c { (cfg_descr == 4) -> (size_decoder ==
5);};
```

Figure 1. Inter-dependent properties

Due to the large interdependency of certain properties in a class, it becomes increasingly difficult for the user to convert the constraints into procedural code maintaining the same valid dependencies between them. Hence such constraints are better solved in parallel by the constraint solver rather than by the verification engineer.

Figure 2 is an example with complex constraints, where '*jd_D.mode'*, '*rate'* and '*rep'* are inter-related with lot of dependencies. If the order of solving these properties is decided by the verification engineer, all the dependencies involved must be taken care manually. When large numbers of such interdependent properties are used segregation of such constraints to procedural and parallel code becomes difficult.

```
class turbo_class;

//property declarations
rand mode_e mode;
rand int rate;
rand int blk_size;
…….
//constraints of properties
constraint rate_c {   (mode == MAX_MODE) -> (rate
inside {0,2,3,4,6});
};

constraint blk_size_c {
if(mode == MAX_MODE && rate == 0)  (blk_size/2
inside
{108,120,144,180,192,216,240,480,960,1440,1920,2400}
) ;

if(mode == MAX_MODE && rate == 2)  (blk_size
inside {48,96,144,192,240, 288, 384, 432, 480}) ; };
constraint s_c {
  ((rep == 3) || (rep == 1))-> mode == TE_MODE; };

constraint s_rm_c {
  (so == 1) -> ( byp != 0);};

constraint dyn_stop_c {
  (dyn_stop == DYN_STOP) -> (so == 0);   };

//functions
…….

endclass
```

Figure 2. Example for complex constraints

In the example shown in **Error! Reference source not found.** there are two random fields, *size_decoder* and *cfg_descr* in class *packet*. When an instance of *packet* class is randomized, these fields get randomized in a bidirectional way based on the constraint specified. User can add more constraints on these fields by extending the *packet* class. When the variable *size_decoder* is independent and *cfg_descr* is dependent on *size_decoder*, generating the fields sequentially will reduce the constraint solver overhead. This can be achieved by generating *size_decoder* during randomize() and *cfg_descr* during post_randomize() methods. This way, the constraint solver needs to solve only *size_decoder* during randomization instead of being burdened by solving both variables. But the side effect of this method is that, user cannot add constraints externally on the field *cfg_descr* which gets generated during post_randomize() call.

Constraints are segregated when it is easier to partition them and there is overhead on constraint solver solving them parallel. But if the constraints are as simple as shown in **Error! Reference source not found.**3, then it is better to generate the random fields in traditional randomization since the overhead on constraint solver is minimal.

```
class packet;

 rand bit[5:0] size_decoder;
 rand bit [3:0] cfg_descr;
 constraint size_c { (size_decoder == 3) -> (cfg_descr
== 1); }
....
endclass
```

Figure 3. Constraint Dependency

In the example shown in Figure 4, '*turbo_pkt_q*' is a queue of instances of class '*turbo_class*' (described in Figure 2). The size of the queue is random and is controlled by the field *q_size*. Constraint block *cst_list* has constraints among the instances of the queue. During randomization, not only the constraints within the turbo_class needs to be solved, but also the constraints among the instances in parallel for the same randomize() call. As the *q_size* increases, the overhead on the constraint solver increases heavily slowing down the simulation and might even result in constraint timeout.

```
  rand bit[5:0] size_decoder;
 rand int q_size;
 rand turbo_class turbo_pkt_q[$];

constraint cst_list {
    foreach (turbo_pkt_q[i] {
      (i!=0) ->
        turbo_pkt_q[i].mode == turbo_pkt_q[i-1].mode;
      (i!=0) ->
        turbo_pkt_q[i].start_addr > turbo_pkt_q[i-
1].end_addr;
   }
}
```

Figure 4. Queue of class handles

Even though the size of the queue is controlled by the field '*q_size'*, maximum number of '*turbo_pkt_q'* instances need to be initialized and pushed into the queue since objects will not get initialized automatically during randomization. That means, for every randomization, constraint solver has to solve all the constraints for the list of maximum size.

Instead of generating all the objects of the queue in parallel, if the objects are generated one by one in the post randomize method incorporating the constraints between the objects procedurally, then the overhead on the constraint solver can be minimized and any number of objects can be generated. In this case, user cannot add constraints to the queue from the test case directly. However, facilities have to be made to add constraints to the object itself through factory mechanism and unique ids of the objects. Please refer the diagram of hierarchical class modeling for procedural solving of constraints in Figure 5.

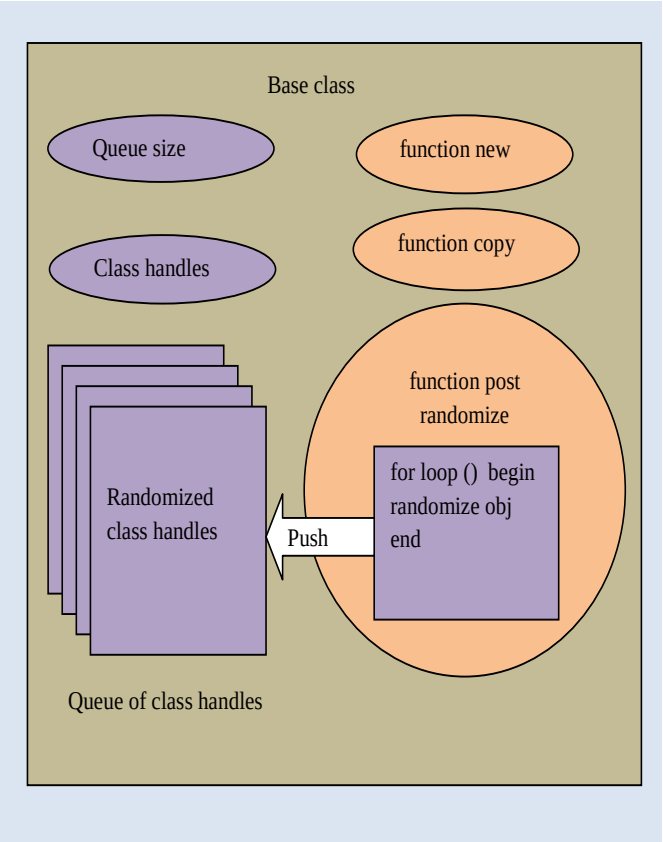

Figure 5. Base class for hierarchical data structure

The partitioning of the procedural and parallel constraints requires in-depth knowledge of the various properties of the classes and their functionality.

While developing the infrastructure for partitioning the constraints, we also found the need for similar data structures in other verification modules. For example, similar to the data structure handling in '*turbo_pkt'*, another type of packet '*viterbi_pkt'* is required for 'viterbi list' generation. Thus for all designs which require similar hierarchical data structure, a generic base class was developed. The section given below on reusable hierarchical class for sequential randomization explains the usage of reusable base class for different types of list generation.

1) Reusable hierarchical class for sequential randomization

In a verification environment with complex packet based protocols, multiple packets need to be generated randomly and inserted into the channel of the signal drivers. This driver in turn drives the packet as per protocol on the bus. In such cases, the configuration such as "number of packets" needs to be generated randomly first and then the individual data packets are randomized using the additional constraints passed from the test case. Each randomized packet is then pushed into a queue. Figure 6 shows the code snippet for the hierarchical class.

```
class base_class #(type child_pkt = vmm_data)
extends vmm_data;
     rand int q_size; //size of queue for class handles
    constraint q_size_c { q_size[31] == 0;};

   child_pkt pkt_obj_q[$];
   rand child_pkt pkt_obj;


  virtual function void post_randomize();
        pkt_obj_q.delete();
        $cast(pkt_obj, pkt_obj.copy(pkt_obj));
    for(int i = 0;i < q_size ; i++)      begin
        ……..
       pkt_obj.randomize();
        pkt_obj_q.push_front(pkt_h);
       ……..
    end
  endfunction
endclass

program test();
  initial begin
     base_class #(turbo_pkt) tr;
     turbo_pkt tpkt;
     tr.pkt_obj   = tpkt; // assigning constrained value
// to baseclass handle
     ……..
     tr.randomize();
   end
endprogram
```

Figure 6. Base class for hierarchical data structure

Generating many random packets after solving large set of parallel constraints to fill the queue puts overhead on the constraint solver engine. Randomization is partitioned into 2 phases to lessen the burden on the constraint solver. However, these data structures involving hierarchical classes have to be coded in every test bench for different protocols which requires rework. Also the probability of error is high since it involves handling of multiple class handles and their randomizations.

For all designs which require similar hierarchical data structures, a new parameterized base class is added to the data class library which is customized by the user according to requirement. The base class has a property which is a class handle of parameterized type, which in turn is used as a factory pattern for the generation of list. The size of the list depends on the random configuration of the base class. Extension of the parameterized sub-classes in test case enables constraining the properties of the class depending on the packet protocols. This helped in achieving both performance and controllability of the transaction from the test case level.

Figure 7 provides a code snippet which instantiates generic base class for generation of '*dec_turbo_enabled_turbo_pkt*' list and also shows use of data_id to control individual object of list.

```
class dec_turbo_enabled_turbo_pkt extends
dec_turbo_job_desc_pkt;

    constraint dec_turbo_enabled_turbo_pkt_c {
        if(data_id) == 1{
          turbo_jd_D == 3;
        }
             };
……

  virtual function vmm_data copy (vmm_data to = null);
      dec_turbo_enabled_turbo_pkt tc;
…….
   endfunction
endclass

program test();
  initial begin
     base_class #(dec_turbo_enabled_turbo_pkt) tr;
     dec_turbo_enabled_turbo_pkt tpkt;
     tr.pkt_obj   = tpkt; // assigning constrained value to
                          //baseclass handle
     ……..
     tr.randomize();
   end
endprogram
```

Figure 7. Generic Base class Code snippet

Care should be taken in copying the randomized handles from one to other before pushing into the queue to avoid pushing the same handle repeatedly. Also constraining individual packet was a problem which is solved by using a

tag named "*data_id*" where the data_id is used to control each packet constraint from the test case.

B. Managing large list size

In verification, maintaining large amount of expected data for each transaction becomes tricky. For example let's consider a simple DMA design which reads data from source location and writes that data in destination location. The number of bytes read and written can be of huge size. For example, in our DMA design, number of bytes read and write depends upon 3 fields in the DMA descriptor/instruction. Each of the 3 fields is 16 bit field. Hence, a total 2^16 * 2^16 * 2^16 = 2^48 bytes can be transferred.

A reference model was developed to generate the expected data/address list which is stored in respective queues. When DUT starts reading and writing data, Scoreboard compares the actual address and data with the expected address and data from the queues in the reference model. Though the idea looks fine and simple there are system memory limitations involved. The amount of data can increase the queue sizes to a value which is beyond the available run time memory.

In our system we observed that when queue size grows and memory usage reaches around 2GB, simulation crashes because of memory unavailability. Though different systems will have different amount of allocated run time memory but beyond a point all will reach their upper memory limit.

Thus to avoid memory limitation 'watermark' concept was adopted. To give a simple analogy, it is just like a water tank where once water reaches a certain lower level, pump starts automatically to fill the tank and once water reaches a certain upper level pump gets auto cutoff. It waits to start again when water reaches the lower level due to consumption and the cycle goes on. The concept is explained part of Figure 8 and 9.

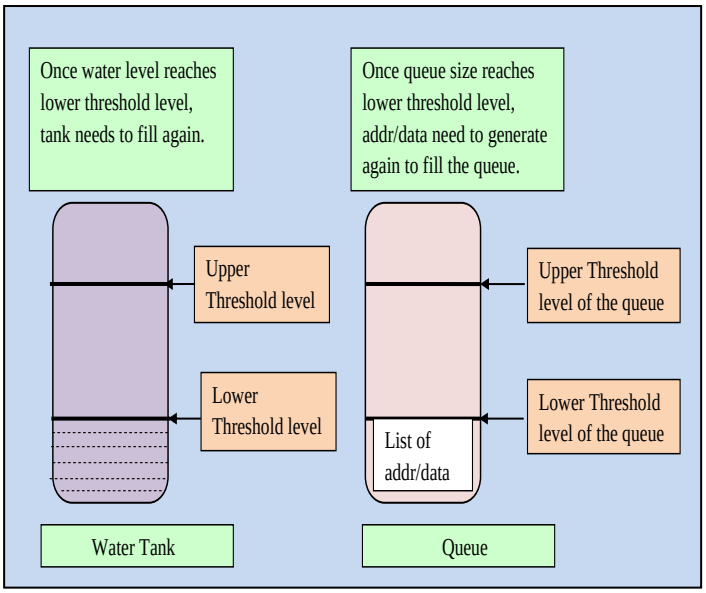

Figure 8. Water Mark Concept: Lower Threshold Level

Similar concept is used in managing the queues in which reference model dynamically generates addresses/data in small chunks and fills those addresses/data in their respective queues. These generated address/data are filled only up to a high threshold level (user can control) in the queue and once the DUT transfers those addresses/data, Scoreboard compares actual data with the expected list and deletes the data from the list. Once it reaches a low threshold level, new chunk of address/data gets generated and again the queues get filled up. This way, all the address/data for each transaction gets generated without affecting the runtime memory.

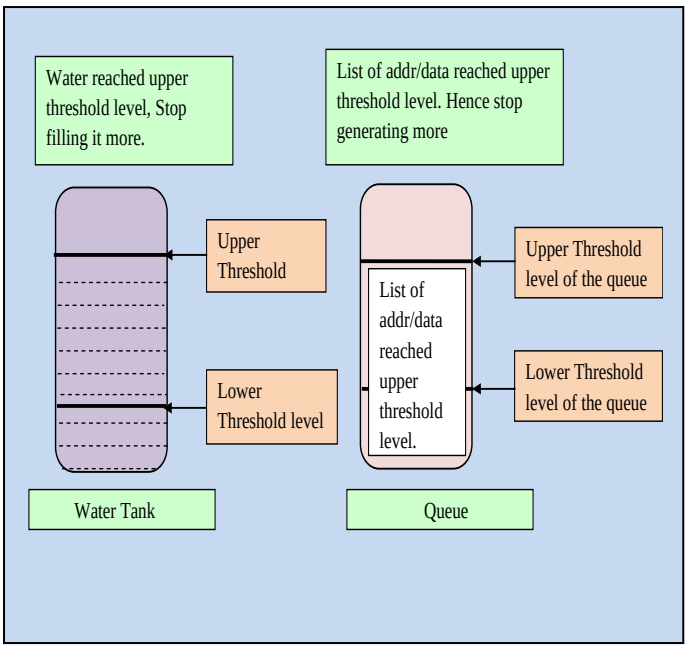

Figure 9. Water Mark Concept: Upper Threshold Level

In our project we have set the lower threshold level for the queue size as 256 and upper threshold level as 1024. We set these values mainly based on the max burst size possible in our design to fetch data at a time and to keep extra buffer such that Scoreboard, DUT overlap does not occur i.e. Scoreboard always has data available before it calls compare method. As we set only 1024 as upper threshold value, it will work smoothly in most of the system. User can set any other values based on their design or available runtime memory.

The same concept is applied in generating the packets as well to avoid generating huge number of random packets and storing them in a queue upfront. With this model, more packets get generated once lower threshold level is reached. Each packet gets generated after solving many constraints. Hence solving constraints of many packets at the beginning will take lot of time and eventually will give constraint time out error at some point. Using watermark concept we were able to avoid this issue as well.

C. Interface independent VIPMaintaining the Integrity of the Specifications

In most designs, the interface protocol is upgraded for better performance. Also the driver interface for configuring the registers keeps changing from block level to chip level. In such cases, the development of the verification IP is impacted since the engineer has to rework on most components like command layer protocol monitors and data handler for every change in the interface as per architectural changes. It also becomes necessary to maintain multiple sets of VIP for different interfaces. Only thing which can be reused is reference model or data integrity logic on the extracted data received from monitors. Thus to make it generic or enable reuse at all verification levels with different modes, a high degree of configurability has to be provided to the verification IP.

For example, in DMA verification environment, the functional driver initializes the descriptor memory space with descriptors and then enables DMA DUT to start the transactions. In module level, DUT is interfaced with internal proprietary interface (Data Bus A as per Figure 10), but in Sub System Level the interface is different (Data Bus X as per Figure 10).When leveraging the VIP at multiple levels, the functional driver's functionality remains the same, but formation of data which is passed to signal layer driver connected to DUT interface needs to be changed. This will need an update to functional driver to change the format of the data as per new interface attached.

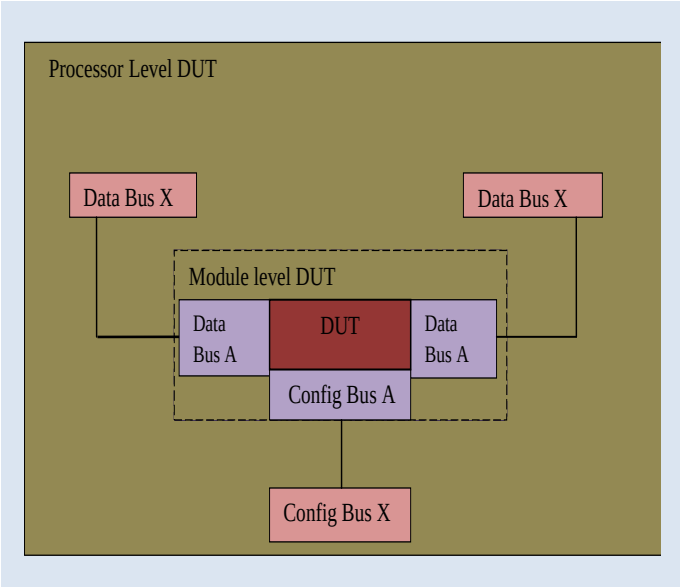

Figure 10. Design Under Test

Also there were more chances of interface getting changed as per performance requirement to support more bandwidth. Thus our VIP design must accommodate these changes to provide reuse and support different configurations as per architecture requirements. Similar issue exists with scoreboard for interface dependent data received from Dut for comparison.

To minimize the impact of frequent specification changes at the interface level, interface driver and its transaction class are separated from the rest of the VIP components. To implement this, a two step approach is followed, a functional level VIP transaction data is created which will interact with rest of the VIP components and a conversion function is provided to convert VIP transaction to interface level transaction. By adopting this method, for any change in the interface level protocol, only the associated driver, its interface level transaction and the conversion function have to be changed. The rest of the VIP components remain unaffected and unchanged.

With new approach DMA verification IP was designed to follow the layered approach specified by VMM methodology. The functional layer consists of reference model, scoreboard and functional driver. The command layer driver/monitor communicates between the functional layer and signal layer components. The information passed to command layer need to be passed to reference/scoreboard model to compare with the actual transactions received from the DUT. The challenge is that the functional driver must have the knowledge of the protocol used to communicate to command layer. Similarly Scoreboard will also need to generate the expected data list as per the protocol data format to be compared with the actual data received from the command monitor.

In order to make the functional layer independent of interface data a two step process is followed:

1. Create the module related data dma_burst_data to be used by functional driver/reference model/scoreboard.

2. Have a converter class to convert from interface (BusA/BusX data) to module specific data class (dma_burst_data) or vice versa when driving data to command layer which is specific with respect to DUT.

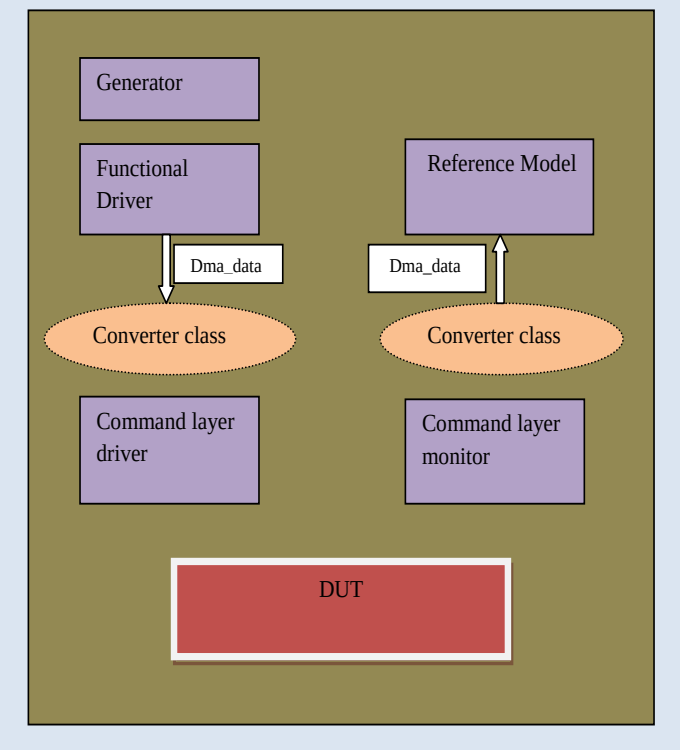

Figure 11. DMA Testbench with Interface independent VIP

Thus with this approach DMA VIP is separated from interface related components and the command layer uses the converter class output data to communicate to DUT. Figure 11 explains how converter class acts as a mediator between functional layer and command layer. Figure 12 is an example of converter class code snippet.

```
// Convert BUSA data to dma _data and
// passed to SB to comapare with expected list

function tr_post_mon_call(BusA_cmd_monitor
xactor,BusA_pkt pkt_mon);

dma_burst_data out_burst;
out_burst = new();

for(int =0;i<=pkt_mon.BusA_burst_size;i++)  begin
   dma_transfer_data dma_tr_temp;
   dma_tr_temp = new();
   dma_tr_temp.address =  pkt_mon.BusA_addr[i];
   dma_tr_temp.data = pkt_mon.BusA_data[i];
……..
……..
……..
end
```

Figure 12. Interface converter class

III. RESULT AND CONCLUSION

- For DMA descriptor list generation similar hierarchical class modeling was required thus reused the generic parameterized base class passing dma descriptor class handle. This helped us not to recode the same data structure.

- Using watermark concept we avoided memory crash issue. Without using water mark concept tool runs out of memory when queue(32bit queue type) size reaches around 149314080 value

- In the course of the project, the processor interface protocol upgraded from AHB to AXI. As the AXI VIP provided the command layer monitor, only effort was to write a conversion method to convert VIP specific transfer to AXI. This took us just half a day effort where as in normal approach at least 3 days to modify functional driver and scoreboard

The table in Figure 12 shows the reduction in time taken by following sequential randomization of the class properties compared to parallel randomization.

No of Pkt	CPU time in seconds in Parallel randomization	CPU time in seconds in Sequential randomization
50	8	2
100	19	2
500	86	8
1000	190	18
4000	788	197

Figure 12. Time - Parallel vs Sequential randomization

The table in Figure 13 shows the reduction in memory consumed by following sequential randomization of the class properties compared to parallel randomization.

No of Pkt	memory in Mb in Parallel randomization	memory in Mb in Sequential randomization
50	16	16
100	16	16
500	16	556
1000	273	873
4000	465	2788

Figure 13. Memory - Parallel vs Sequential randomization

Following above recommendations, a robust VIP is created to generate random packets which are constrainable and easily controlled from test case without burdening the constraint solver. This reusable VIP is efficient, faster, and immune to interface changes. Also reduces run time memory consumption.

IV. ACKNOWLEDGEMENT

We would like to thank LSI and DVCon for giving us the opportunity to present our work and share our experience in a diversified forum. We also express our gratitude to Dwaraka Jayendra, our manager for his guidance and help.

V. REFERENCE

[1] IEEE Standard for System Verilog - Unified Hardware Design, Specification,and Verification Language

[2] Verification Methodology Manual for SystemVerilog by Janick Bergeron, Eduard Cerny, Alan Hunter and Andrew Nightingale