Using Advanced OOP Concepts To Integrate Templatized Algorithms for Standard Protocols With UVM

Anunay Bajaj
Synopsys India Pvt. Ltd.
New Delhi
0091.11.49233971
abajaj@synopsys.com

Gaurav Chugh
Synopsys India Pvt. Ltd.
New Delhi
0091.11.49233980
cgaurav@synopsys.com

Abstract: Reusability is no longer a nice-to-have requirement when it comes to the development of a successful verification environment for a System On Chip(SoC) that incorporates multiple standard protocols. The more reusable the code and the implementation of verification IPs, the faster will be its development and verification cycle. From an organizational point of view, the major time and effort penalty that one has to pay is writing and verifying the same set of code repetitively across multiple projects.This is one of the prime hidden factors in the latency of Project Development to Delivery Cycle.

This paper emphasizes the incorporation of many standard encoding,data integrity, and common protocol intersection schemes to become a part of the Universal Verification Methodology(UVM) library package through the concepts of specialized container classes. To support the dynamic and ever-upgrading nature of verification industry, several template class packages for common layers across multiple protocols and complex mathematical operations are proposed to achieve effective reusability and universally-accepted standard-quality product across the entire verification industry.

Keywords
UVM,Encoding,Decoding,Clock,Package,Digital SignalProcessing,SoC,LaneManagement,Filters, Sampling

I. INTRODUCTION

The *Time is money* adage stands true in the current cut-throat competition. Why would one want to reinvent the wheel? The same stands true in the field of Design and Verification of SoC; the man-hours put in to implement a commonly-used logic or an already implemented block with a few minor application-specific changes is one of the prime unknown time and effort penalty that one has to pay without realizing the fact that the same task could be implemented in a much smarter way.

II. NEW AGE METHODOLOGY: UVM WITH A DIFFERENCE

We all agree that methodologies in the field of SoC verification has rendered the life of a testbench developer and an end-user much simpler.The efficient idea of introducing methodologies is to have a standard layout across all the Verification IPs. A well-defined structure helps even a naive developer to get an overview of the flow control of any testbench architecture.However, the current standard methodology, that is, UVM can be armed to become more protocol friendly and developer aware.

The proposal in this paper is to empower UVM with technology-independent and developer-oriented container classes, which will be generic in nature and flexible enough to be molded according to the type and requirement of a testbench creator.

SystemVerilog with its advanced Object-Oriented Programming (OOP) concepts along with UVM can be harnessed to provide a common and a standard platform to developers. They could get a feel that their job is already partially done and they just need to map or override these containers to suit their verification architecture.

There are many coding algorithms, also termed as schemes, which are often used across multiple verification environments in the industry. These could be standardized as a part of the standard UVM package. This paper throws light on the categories and mechanism of implementation of these algorithms. However, this does not mean that the proposed mechanisms are the only ways to implement; more efficient and flexible ways can be evolved.

III. COMMONALITIES ACROSS PROTOCOLS

We shall now look at some of the common features across various protocols, discuss how to harness these features, and make a part of UVM.

A. Encoding/Decoding and Encryption Algorithms

These algorithms are often implemented and exhaustively used in PCI Express,SATA,Ethernet,and Interlaken, etc. One of the most common algorithms is *8b/10b Coding*.

Encryption algorithms across wired and wireless protocols, for example Advanced Encryption Standard (AES) in Bluetooth, seem ubiquitous.

B. Data Integrity

To maintain data integrity through error-detecting codes, find extensive usage in digital networks and storage devices to detect accidental changes to raw data is a mandatory block. In most of the cases based on a polynomial division, data corruption is monitored. Cyclic Redundancy Check (CRC), Parity checks, and checksum calculations are some of the most abundantly used data-integrity checking methods.

C. Digital Signal Processing (DSP)

Many wireless, audio, and video-codec blocks in SoCs require complex mathematical operations. In addition, designer-customized blocks for instance, the circuits where standard hardware multiplier and transform blocks are replaced with typical application-specific block find a lot of Digital Signal Processing (DSP).

D. Clock Generation and Recovery

This is the most common block across all the SoC. In high-speed serial data streams where clock is not accompanied, clock recovery blocks come handy. Also, in slower data-rate protocols, such as MIPI-HSI, where optimality of design is prerequisite, clock is recovered based on the data obtained at the receiver end along with the status of some control signals or reference clock. Here, reusability of these designs is an important feature.

E. Data Sampling Algorithms

Video coding and wireless protocols use a host of sampling techniques, such as Phase Shift Keying (PSK), Manchester Coding, and Non-return-to-zero (NRZ), etc. Even in a Bus protocol, such as Universal Serial Bus (USB), implements the NRZ coding exhaustively.

F. Lane Management

Communication protocols, such as MIPI, Interlaken, and PCI Express, etc. where data streams or packets are sent across multiple lanes, the algorithms for the management of these protocols have similarities.

IV. INTELLIGENT UVM:MON AMI

This paper proposes three distinct packages implementing the commonalities across various designs so that the development of verification environment for the same becomes a much more easier, effective, and comprehensive in nature. These packages, if become a part of the standard UVM package, can really propel it to be stated as a new age methodology.

These packages broadly include template classes also termed as containers for the following components:

- Data Conversion algorithms
- Clock Generation and Recovery blocks
- Complex Mathematical operations
- Data Sampling
- Data Integrity
- Running Disparities
- Lane Management

Table I. gives an overview of the package classification along with their supported templates.

TABLE I. PACKAGE-SUPPORTED TEMPLATES

Package Name	Supported Templates
Standard **T**emplate **A**lgorithms **C**ontainer (**STAC**) classes	1) Common Encoding/Decoding algorithms 2) Data Integrity algorithms 3) Linear Data algorithms
Standard **T**emplate **M**ath **C**ontainer(**STMC**) classes	1) Standard and Complex Mathematical operations for Dynamic computer and mathematical programming-specific RTL cores 2) Digital Signaling algorithms

Standard **T**emplate **U**tility **C**ontainer (**STUC**) classes	1) Clock-related utilities, such as its generation and detection 2) Data Sampling algorithms 3) Lane Management

These proposed template classes can be included as a part of UVM on per-need basis depending on the developers' extent of usage. These classes wrapped in packages can be included in a chained manner along with the UVM package as:

import uvm_pkg::;*
import uvm_<package name>_pkg::;*

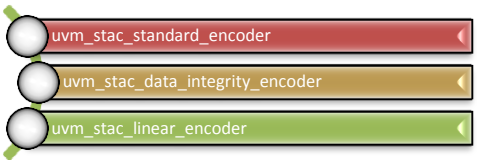
It is important to note that these packages are very exhaustive and contain almost all the algorithms if they go on to become a part of UVM. Here, we will discuss with a bird's-eye view on how the packages and their containers would have a look and feel.

V. PROPOSED PACKAGES

We shall see the pseudocode or skeleton of the Template classes packages in this section.

A. Standard Template Algorithms Container (STAC) Classes Package

package uvm_stac_pkg



1) *Common Encoding/Decoding Algorithms*

Encoding/Decoding and other line coding algorithms to optimize the bandwidth usage and achieve DC balance form an integral part of almost all standard and customized blocks of SoC designs.

Similarly, with the STAC package up and running, testbench developers only need to identify the required blocks of an algorithm based on their needs and can use it umpteen times.The templates of this package are flexible enough to customize the encoding/decoding blocks as per needs.

Some of the frequently used algorithms are 8b/10b; 64b/67b; and 128b/130b, etc. These schemes are extensively used in Ethernet, PCIe, USB3.0, Interlaken,and many other communication protocols.

The *uvm_stac_standard_encoder* class provides a free hand to users to pass the encoding scheme through the ENCODING_TYPE variable with the input, termed as INSTREAM. The *standard_encode_data* function operating as the code-conversion machine,in turn, calls an algorithm-specific function, which converts and returns the output, termed as OUTSTREAM. The function is virtual, thus, can be overridden whenever and wherever a developer wants. INSTREAM and OUTSTREAM are designed to support dynamic arrays in nature to cater any width of data.This code-conversion machine's salient driving factor is the encoding type scheme that users provide.Its internal case statement takes the ENCODING_TYPE *_e_type* as an argument and calls the relevant function to code or decode INSTREAM *_in_data* and return OUTSTREAM *_out_data*.

The standard UVM document will maintain a table containing all the encoding/decoding algorithms against their parameter names that will be passed as an argument in *standard_encode_data*,any other data sampling, or conversion algorithm. The UVM document (User's Guide) table with a few of the coding schemes will look as below:

TABLE II.CODING SCHEME VS PARAMETER NAME

SNo.	Coding Scheme	Parameter
1	8b/10b	`8B10B
2	n-Bit Cyclic Redundancy Check	`NBITCRC
3	Manchester Coding	`MANCHESTER
4	Digital Butterworth Filter	`FILTER

Figure 1 shows the pseudocode of *uvm_stac_standard_encoder*.

```
class uvm_stac_standard_encoder
      #(int    ENCODING_TYPE=0,
        type INSTREAM=bit,
        type OUTSTREAM=int
       )extends uvm_object;

      int _e_type = ENCODING_TYPE;

      virtual function bit standard_encode_data
        (input INSTREAM _in_data,
         output _out_data
        );
        case(_e_type)
          `8B10B_CODE:
            f_8b_10b(_in_data,_out_data);
```

```
        `128B130B_CODE:
          f_128b_130b(_in_data,_out_data);

        `64B66B_CODE:
          f_64b_66b(_in_data,_out_data);
          /* -------------------------
             More Standard Encoding/Decoding
             Schemes in the code
            ------------------------- */
         /*-------------------------
            Provision for User Created
            Schemes Below
            ------------------------- */
      `USER_DEFINED_CODE:
          f_user_coding_scheme(_in_data,_out_data);
   endcase
endfunction
 /* -------------------------
   Generic Conversion Functions
   Definitions for all the algorithms
    ------------------------- */

   virtual function void f_8b_10b
      (input INSTREAM _in_data,
       output OUTSTREAM _out_data
      );
          // Input  : 8-bit  data
         // Output : 10-bit encoded data
 endfunction

 virtual function void f_128b_130b
   (input INSTREAM _in_data,
    output OUTSTREAM _out_data
   );
       // Input  : 128-bit  data
      // Output : 130-bit  encoded data
endfunction
```

Figure 1: Common Encoding/Decoding Algorithms

2) Data Integrity Algorithms

Data integrity algorithms are paramount for a protocol as they maintain the accuracy and consistency of data sent across layers or links.The STAC package provides built-in data integrity creation and verification functions wrapped in the *uvm_stac_data_integrity_encoder* class.This class contains an extra parameter, ENCODE_POLYNOMIAL, and remaining parameters are same as *uvm_stac_standard_encoder*.

The ENCODE_POLYNOMIAL parameter provides a space to a testbench developer to input the custom or protocol-specific coefficients of polynomial.

The skeleton of this class be seen in Figure 2.

```
class uvm_stac_data_integrity_encoder
      #(int  ENCODING_TYPE=0,
        int  ENCODE_POLYNOMIAL=1,
        type INSTREAM=bit,
        type OUTSTREAM=bit
       )extends uvm_object;

      int _e_type = ENCODING_TYPE;
      int _e_poly = ENCODE_POLYNOMIAL;
```

```
       virtual function void encode_data_integrity
         (input  INSTREAM _in_data,
          outputOUTSTREAM _out_data
         );
         case(_e_type)
           `NBITCRC:  f_n_bit_crc
              (_e_poly,_in_data,_out_data);

           `ODDPARITY:f_odd_parity
             (_in_data ,_out_data);

           `EVENPARITY:f_even_parity
             (_in_data,_out_data);

           `USER_DEFINED_CODE:
                 f_user_coding_scheme
             (_e_poly,_in_data,_out_data);
                //More Coding Algorithms
       endcase
     endfunction

/*The implicitly called functions in the case statements
have their definitions separately in the same class*/
Here is how they look:

          virtual function void f_n_bit_crc
            (input ENCODE_POLYNOMIAL _e_poly,
                   INSTREAM _in_data,
             output OUTSTREAM _out_data
            );
               // Input_1  :  Polynomial
              // Input_2  : Input Data
             // Output   : N-Bit CRC value
          endfunction

          virtual function void  f_user_coding_scheme
            (input INSTREAM _in_data
             output OUTSTREAM _out_data
            );
              //User Code Here...
          endfunction

endclass// uvm_stac_data_integrity_encoder
```

Figure 2:Data Integrity Algorithms

The steps for a developer to use the *uvm_stac_data_integrity_encoder* template class in the verification environment are as follows:

(i) Specialize the *uvm_stac_data_integrity_encoder* template class with appropriate values and types for ENCODING_TYPE, ENCODE_POLYNOMIAL, INSTREAM, and OUTSTREAM.
(ii) Create an object of a specialized class.
(iii) Call the *encode_data_integrity* function by providing an input bit stream,*_in_data*.
(iv) The return data stream from the function gives an output bit stream,*_out_data*.

Parity calculation can be done in the same way

Users can also create customized data-integrity algorithms by overriding the *f_user_coding_scheme* function.

3) Linear Data Algorithms

The *uvm_stac_linear_encoder* template contains some of the very important and typical communication-coding techniques.
For communication protocols, where the data or signal encoding and decoding process is continuous as well as repetitive in nature, this template would be very handy.

The skeleton of this class be seen in Figure 3.

```
class uvm_stac_linear_encoder
    #(int   ENCODING_TYPE=0,
      int   BLOCK_LENGTH=0,
      int   MESSAGE_LENGTH=0,
      type  INSTREAM=bit,
      type  OUTSTREAM=int
     )extends uvm_object;

     int _e_type = ENCODING_TYPE;
     int _b_length = BLOCK_LENGTH;
     int _m_length = MESSAGE_LENGTH;

     virtual function void encode_linear_data
        (input INSTREAM _in_data,
         output OUTSTREAM _out_data
        );

       case(_e_type)
       `HAMMING: f_hamming_code
                    (_in_data,_out_data);

       `REED_SOLOMON: f_reed_solomon_code
                    (_in_data,_out_data);

            // ..... Other cases here
     endcase
   endfunction

   virtual function void f_hamming_code
      (input  INSTREAM _in_data,
       output OUTSTREAM _out_data
      );

    /*Takes Input data and depending upon the
      Block and Message length returns Hamming
      Code Output*/
   endfunction

   virtual function void f_reed_solomon_code
      (input  INSTREAM _in_data,
       output OUSTSREAM _out_data
      );
       // Implementation same as above
   endfunction

endclass //uvm_stac_linear_encoder
```

Figure 3:Linear Data Algorithms

Let us take the example of the Hamming code.
The *f_hamming_code* function returns data streams by just passing BLOCK_LENGTH, MESSAGE_LENGTH, and INSTREAM as inputs.

B. Standard Template Math Container (STMC) Classes Package

It is difficult to think about any SoC without DSP blocks as an integral part of it, especially when Video and Audio decoding algorithms are vastly implemented in them. The STMC package contains template classes for all the complex mathematical operations often hand in glove with DSP algorithms. This package acts as a wrapper around Direct Programming Interface (DPI) calls to C-programming language

package uvm_stmc_pkg

1) Z-Transform

The equation of Z-Transform is:

$$X(z) = \mathcal{Z}\{x[n]\} = \sum_{n=-\infty}^{\infty} x[n]z^{-n}$$

$$z = Ae^{j\phi} = A(\cos\phi + j\sin\phi)$$

Figure 4:Z-Transform

Here, developers include the *uvm_stmc_pkg* package in their environment and use the *uvm_stmc_math_encoder* container class.The PHASE _phase, TIME_LOWER_LIMIT _t_ll, and TIME_UPPER_LIMIT _t_ul values are required for a typical z-transform of any digital information spread across a time bandwidth.The *f_z_transform* function takes these overridden values specified by developers and returns the bit stream of transformed variables.Wherever calculations of complex equations are required, the DPI call to C will be implicitly called by the function itself to return the desired output.

2) Digital Filters

The realization of Digital Filters for the Core Digital Signal and Image processing blocks that require verification can be done using STMC.

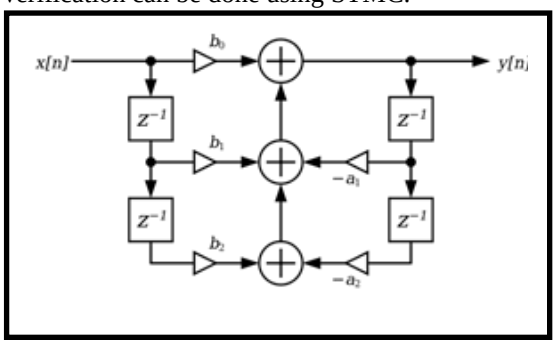


Figure 5: A Digital Filter Example

Filter output:
$y[n]=b_0x[n]+b_1x[n-1]+b_2x[n-2]-a_1y[n-1]-a_2y[n-2]$
n is a timestamp or time instant,x components are inputs, and y components are outputs with their respective $a_{var}$and b_{var} coefficients.
Based on the time range captured, any standard or customized filter can be implemented just by providing the order and summation of filter inputs based on the time range can be captured and converted using the STMC template.

The skeleton of this class be seen in Figure 6.

```
class uvm_stmc_math_encoder
      #( type   INSTREAM=bit,
         int    ENCODING_TYPE=0,
         int    ORDER=0,
         int    PHASE=0,
         int    POLES=0,
         int    ZEROS=0,
         int    TIME_LOWER_LIMIT=0,
         int    TIME_UPPER_LIMIT=0,
         type  OUTSTREAM=bit
        )extends uvm_object;

      int _e_type = ENCODING_TYPE;
      int _order = ORDER;
      int _phase = PHASE;
      int _poles = POLES;
      int _zeros = ZEROS;
      int _t_ll = TIME_LOWER_LIMIT;
      int _t_ul = TIME_UPPER_LIMIT;

      virtual function void math_encoder
       (input INSTREAM _in_data,
        output OUSTREAM _out_data
       );
       case(_e_type)
         `Z_TRANSFORM:
            f_z_transform(_in_data,_out_data);

         `FILTER:
            f_filter (_in_data,_out_data);

               // ……. Other cases here
       endcase
     endfunction
endclass //uvm_stmc_math_encoder
```

Figure 6: Math Encoder

The steps for developers to use the *uvm_stmc_math_encoder* template class in verification environment are as follows:

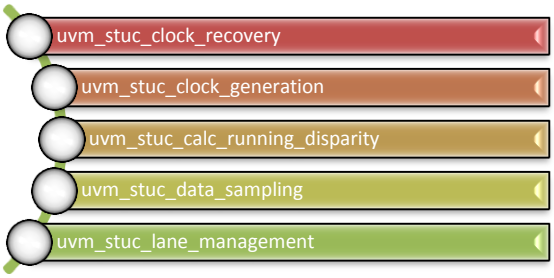
(i) Specialize the *uvm_stmc_math_encoder* template class with appropriate values and types for INSTREAM, ENCODING_TYPE, ORDER, PHASES, POLES, ZEROS, TIME_LOWER_LIMIT,TIME_UPPER _LIMIT, and OTSTREAM.
(ii) Create an object of a specialized class.
(iii) Call *encode_linear_data* by providing input bit stream *_in_data*.
(iv) The return data stream*_out_data* of the internal function *f_filter* will give the desired digitally filtered output.Filter can be Butterworth or Chebyshev in its implementation.

An exhaustive STMC package contains a library of various DSP algorithms and other filters to suit developers' need.

C. *Standard Template Utility Container (STUC) Classes Package*

As the name suggests, this package contains all the utilities common to most of the protocols.
Algorithms, such as Clock generation, Clock Recovery, Clock stretching, Timer, and other clock-related blocks are the part of this package.

package uvm_stuc_pkg



1) *Clock Recovery*

Protocols, such as SATA and MIPI-HSI etc. find a very extensive use of clock recovery, thus, making imperative for this logic to be a standard UVM source code block.
A very basic Clock Recovery block has the edge detection and Sampling Circuitry blocks in it. The Edge Detection block detects all logic 0 to logic 1 transitions of an incoming data and vice versa. This function is just like a Phase Locked Loop (PLL). The output of this block is fed to the Sampling Circuitry block.The Sampling Circuitry block implements a sampling counter,which monitors and samples the edges detected by the previous block and outputs the clock.Refer to Figure 7 for the Clock Recovery Block diagram and Figure 8 for its pseudocode.
The *uvm_stuc_clock_recovery* class of the STUC package has the *uvm_clock_rec* task, which takes data stream *_in_data* and a reference clock *_clk* as an input.

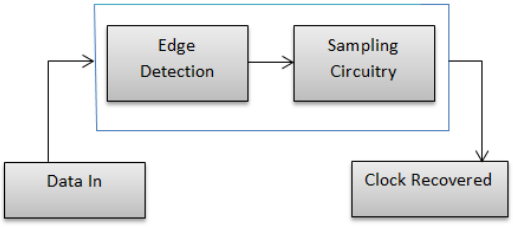


Figure 7: Clock Recovery Block Diagram

```
class uvm_stuc_clock_recovery extends uvm_object;

virtual task uvm_clock_rec
   (ref bit  _in_data,
             _clk,
    ref _out_data
   );

  t_edge_detection_and_sampling_circuitry
 ( _in_data, _clk, _out_data);

/*Logic for Edge detection outputs the edges which is
fed to Sampling Circuitry*/

/*Output Clock from the Sampling circuitry is
recovered with the help of reference clock as the
circuitry's inbuilt sampling counter samples the edges
obtained from the previous block*/

  endtask

endclass //uvm_stuc_clock_recovery
```

Figure 8: Clock Recovery

The *uvm_clock_rec* task wraps the Edge Detection and Sampling Circuitry blocks into a single task, namely *t_edge_detection_and_sampling_circuitry*, and can be overridden. The clock recovery block of verification environment needs to provide the input data stream and a reference clock of developers'choice.

One thing to note here is that this is a very basic implementation of Clock Recovery shown as an instance; but provision for more complex logic by adding more conversion functions can become a part of this class.

2) *Clock Generation*

The Clock generation pseudo code is in Figure 9

```
Top level module
module top;
   bit clk;
   real freq;
   real duty_cycle;

  uvm_clock_generator clk_gen_object;
  clk_gen_object=new();
  initial begin
       clk_gen_object.uvm_stuc_clock_gen
           (clk,freq,duty_cycle);
        //Other user code here
  end
endmodule

class uvm_stuc_clock_generator extends uvm_object;

    virtual task uvm_clock_gen
     (ref   clk,
      ref   freq,
      ref   duty_cycle
      );
      //Clock generation code here
    endtask
endclass //uvm_stuc_clock_generator
```

Figure 9: Clock Generation

The clock generation logic class *uvm_stuc_clock_generator*, is a standard clock generation class with an inherent task *uvm_clock_gen*, which has *clk*, *freq*, and *duty_cycle* as referenced arguments. Developers' top-level module can instantiate the class handle and allocate memory to it. In the verification environment, the *uvm_clock_gen* task is called with the clock variable - *clk* and the custom frequency - *freq*, and the duty cycle - *duty_cycle* as its arguments.

3) *Running Disparity*

On a broad-level, Running Disparity (RD or rd) is the difference between the number of logic 1-bit and logic 0-bit between the start of a data sequence and a particular instant in time during its transmission. The RD for a data stream is the difference between the number of logic 1-bit and logic 0-bit in that stream. For example, if there are more 1 bits than 0 bits, the RD is defined as positive. If there are fewer 1 bits than 0 bits, the RD is defined as negative. If the number of 1 bits and 0 bits is the same, the RD is defined as neutral or zero. Thus, RD helps in achieving the Direct Current (DC) balance.

This important concept is applied across many communication protocols, such as Ethernet,Interlaken,and PCI Express, etc.

The STUC package contains container classes to calculate the RD of the input stream entered by a testbench developer.Figure 10 shows the pseudo code of class *uvm_stuc_calc_running_disparity*.

To calculate the RD of encoded data(eg. 8b/10b), a Lookup table is maintained by the *calc_running_disparity* function. Based on the *_rd_value* input provided from the environment, the *_out_data* will have either positive or negative disparity.

```
class uvm_stuc_calc_running_disparity
       #(type INSTREAM=bit,
         type OUTSTREAM=bit
        )extends uvm_object;

  virtual function  void  calc_running_disparity
   (input INSTREAM _in_data,
    input _rd_value=0
    output OUTSTREAM _out_data
   );
    //Generates _out_data on _in_data and
    //_rd_value representing +/- disparity number
  endfunction
endclass//uvm_stuc_calc_running_disparity
```

Figure 10: Running Disparity

4) *Sampling Algorithms*

Data and clock sampling techniques in Video and Audio codecs are widely used along with many wireless technology protocols.

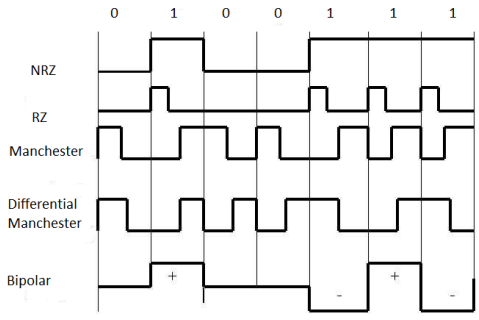


Figure 11: Common Sampling/Encoding Schemes

On the fly sampling of data facility by just mentioning the type of scheme is the purpose of the *uvm_stuc_data_sampling* container class

```
class uvm_stuc_data_sampling
      #(int SAMPLING_TYPE=0
       )extends uvm_object;

       int _s_type=SAMPLING_TYPE;

       virtual task data_sampling
          (ref bit _in_data,
           ref bit _reference_clk=0,
           ref bit  _out_data
          );
          case(_s_type)
           `NRZ:
             t_nrz_code(_in_data, ,_out_data);

           `MANCHESTER:
             t_manchester_code
              (_in_data,_reference_clk,_out_data);

           `PSK:
             t_psk_code(_in_data,out_data);

            //More sampling algorithms
          endcase
       endtask

       virtual task t_nrz_code(input _in_data,output
                                          _out_data);
              //NRZ Code conversion here
       endtask

       virtual task t_manchester_code(input
              _in_data,_reference_clk,output _out_data);
        //Manchester Code conversion here
       endtask

endclass //uvm_stuc_data_sampling
```

Figure 12: Data Sampling

Figure 11 denotes the most common data sampling schemes.

The skeleton of the *uvm_stuc_data_sampling* class can be referred from Figure 12. According to the sampling scheme passed as an argument in the *data_sampling* task along with the input data stream and a reference clock can be done effectively. This reference clock may or may not be required based on the selected sampling scheme. The case statements or the sampling machine inherently calls the code conversion task for the desired SAMPLING_TYPE - *_s_type*.

5) Lane Management

Active lane management from the bundle of lanes is an important feature across MIPI,PCI Express,Interlaken,and Ethernet,etc.

```
class uvm_stuc_lane_management
      #(type INSTREAM=bit,
        int WIDTH=0,
        int LANES=0,
       )extends uvm_object;

       int _active_lanes;
       bit [WIDTH-1:0]  _lane_count [LANES][$];

       virtual function void lane_mgmt
          (input  INSTREAM _indata,
                  WIDTH _width
          );

         case(_active_lanes)
           //Function calling based on the number of
           //active lanes
         endcase
       endfunction
endclass //uvm_stuc_lane_management
```

Figure 13: Lane Management

This is a very important utility to be a part of the STUC package as it reduces a lot of overhead of developers.

In Figure 13, the *uvm_stuc_lane_management* class contains the input data,INSTREAM, which has to be divided across lanes as specified by users in the LANE variable. The WIDTH variable signifies the width of the data to be sent across the lanes.OUTSTREAM is the output data per lane.

The steps for developers to use the *uvm_stuc_lane_management* template class in a verification environment are as follows:

(i) Specialize the *uvm_stuc_lane_management* template class with appropriate values and types for INSTREAM, WIDTH, and LANES.

(ii) Create an object of the specialized class.

(iii) Set the number of active lanes using the handle as follows:
handle._active_lanes=<value>

(iv) Call the *lane_mgmt* function with a developer-specified data vector

(v) Capture the output per lane in the array of the *_lane_count* queue.

VI. SUMMARY

The paper has given one of the many possible ways to implement the commonalities across various standard protocols and application-specific blocks. The only motive to include these packages with multi functionalities is to add more value to the existing UVM environments and to make developers' life easier. The templatized approach minimizes repetition and thus, saves the development and debug time. The packages are forward compatible in nature to cater future requirements and the new as well as the existing verification environment.Developers should feel that they are at an entirely new level and are not developing from the scratch. By making things more industry standard yet flexible, the shift from one block to another becomes easy. The consistency of the architecture and its maintenance is another feature that comes with these packages.

VII. REFERENCES

[1] Robert Lafore,Object Oriented Programming in C++,4/E
[2] IEEE 1800-2009 System Verilog LRM
[3] Universal Verification Methodology (UVM) 1.1 User's Guide
[4]MindShare,PCI Express System Architecture
[5] John G. Proakis,Digital Signal Processing: Principles, Algorithms, And Applications, 4/E