UVM-FM: Reusable Extension Layer for UVM to Simplify Functional Modeling

Ahmed Kamal
Mentor, a Siemens Business
Cairo, Egypt
ahmed_kamal@mentor.com

***Abstract*- One of the main challenges in SoC verification is time-to-market pressure, for that reason all verification engineers are looking for new approaches to speed up building and developing their test environments. Universal Verification Methodology (UVM) is the de facto method as it combines many common approaches to standardize the test environment architecture. However, the increase in modern systems complexity introduced new obstacles such as modeling layered protocols. This paper introduces a reusable backward compatible extension layer for the UVM package (UVM-FM), and changes the UVM agent architecture to a network-like topology. UVM-FM solves the communication challenges in the layered protocol modeling process and speeds-up the UVM environment creation in a timely manner. In addition to that, the proposed extension for the UVM package gives more debugging capabilities, which is an important key element.**

I. INTRODUCTION

Universal Verification Methodology (UVM) [1] is a powerful standardized tool to implement a reusable verification environment, especially for single-layer protocols such a memories and simple communication buses, where a single uvm_agent and uvm_driver are enough to model an exerciser to the design under test (DUT). Layered protocols have become very important key elements in the SoC industry, where the protocol defines several layers that contribute together to perform the overall protocol functionality. Most of the industry modern protocols are designed with this stacked approach, such as PCIe, SATA, SAS, and UniPro. UVM does not standardize how to implement a verification environment for layered protocols, and does not recommend implementing the environment by either using a single agent or multiple agents where each agent represents a layer from the protocol stack. For that reason, several approaches were proposed by different verification engineers to address this issue and suggest a robust solution. Section II explores some of the suggested ideas and approaches before introducing our proposed solution (UVM-FM).

UVM-FM is a reusable extension layer for UVM that simplifies functional modeling for layered protocols; it is backward compatible with the traditional UVM package. The new approach introduces changes to the uvm_agent architecture to be network-like topology; consequently, the communication between different components in the same layer or adjacent layers will be standardized and abstracted with simple subroutines. Section IV provides detailed description for the new agent topology as well as how the different layers will communicate together. Section V provides a detailed description for UVM-FM package, and lists the new components specification. The paper recommends that the UVM-FM solution to be part of the UVM standard; until that happens, the user could use the specifications mentioned in this paper to build a local reusable version of UVM-FM.

This approach is used to implement a verification environment for one of the industrial layered protocols, which is Serial Attached SCSI (SAS) [2]. UVM-FM was used to model the link layer, port layer, and it manages the communication between them. Section VI provides details about this case study and highlights the benefits of using UVM-FM in terms of ease-to-model, ease-to-modify and ease-to-debug.

II. RELATED WORK

Fitzpatric [3] and Doulos [4] introduced a technique to manage the layered protocol modeling by using translator sequences. In this approach, there is a separate sequencer for each non-leaf layer and there are free running translator sequences that manage the communication between each two adjacent layers. Inside the translator sequences, the layer functionality is modeled to translate the upper layer sequence item to the lower layer sequence item. This translation process is replicated until the leaf layer receives a sequence item to exercise the bus. Fig.1 shows how the translator sequences mechanism works in a system that consists of three layer: A, B and C (leaf layer).

This solution solves the problem when the translation from the upper layer to lower layer is straightforward and does not depend on feedback from the lower layer. In the modern SoCs, the layered architecture becomes very complex as it interacts with the lower and upper layers to perform the desired functionality. In addition to that, the layer implementation should take into consideration how to manage downstream and upstream (bi-directional) traffic (messages) at the same time with the lower layer because there might be dependencies between them.

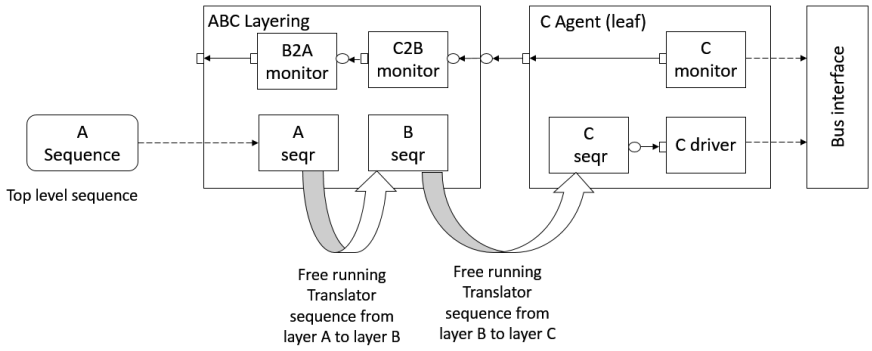

Figure 1. Modeling multi-layers protocols by using translator sequences approach

H.Yu and C.Thomson [5] introduced a similar approach but instead of using translation sequence to convert the upper layer sequence item to the lower layer sequence item and vice versa, they do the conversion inside the sequence item itself and the scoreboard component. They introduced this idea to simplify Fitzpatric and Doulos approach, but their method still faces the same challenges mentioned about the translation sequence method.

All the mentioned approaches try to avoid using R.Chauhan and R.Ganti [6] approach that uses a separate agent for each layer. The complexity of creating such a big environment and managing the communication between the different agents was the motivation of the other approaches to use a single agent to avoid these challenges. UVM-FM accepts the multi-agents approach and introduces a standard communication mechanism between the different layers to ease building the environment. A multi-agent approach will be a key element to solve the synchronization between the different layers, which was a problem in single-agent approaches.

III. MODERN SOC LAYERED ARCHITECTURE CHALLENGES

In modern SoC layered architecture, each layer consists of different blocks working concurrently with independent states to perform different tasks. Fig.2 shows an example for a layered architecture that consists of two layers and each layer has three blocks. In this system, there might be two active communication channels between the two layers, for example, Block A and Block E communicate together, while Block D and Block C communicate also concurrently. These different concurrent communication channels between the different layers are very important to model modern full-duplex serial bus communication protocols.

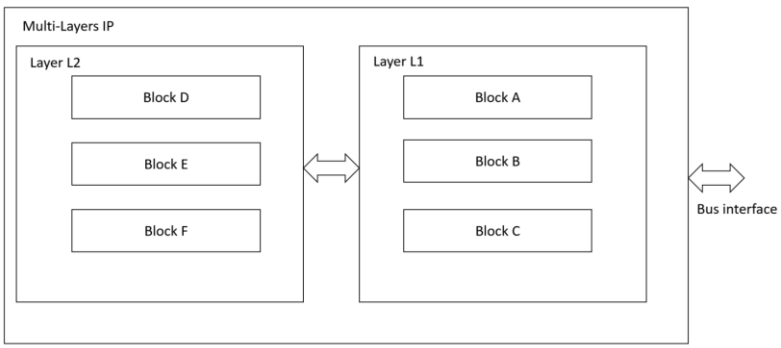

Figure 2. Example for IP stack for multi-layer protocol

IV. THE PROPOSED SOLUTION

In order to overcome the communication challenges between the different layers, this paper introduces changes in the UVM agent architecture to be network-like topology. Agent C in Fig.1 is an example of a traditional UVM agent. The proposed solution introduces a new UVM component, the UVM communicator. This component acts like a router that manages the communication inside the layer and the communication with other layers. Fig.3 shows the proposed architecture changes on the uvm_agent. For non-leaf agents, there are four drivers attached to the communicator node, each driver manages a communication direction with the upper layer or the lower layer. The layer blocks and drivers attached to the communicator node are implemented by using extended versions of "uvm_component" and "uvm_driver", and they are connected to the communicator node by using TLM ports. For leaf agents, it is easy to observe that "To Lower Layer (TLL)" and "From Lower Layer (FLL)" drivers are not needed. The TLL driver is replaced by a component that acts like the traditional UVM driver (converting TLM transactions to signal level), and the FLL driver is replaced by a component that acts like the traditional UVM monitor. The drivers attached to the communicator node are extended versions from uvm_drive.

The new driver and component versions plug easily to the communicator node. The communicator node receives the TLM transaction from any attached node, then it re-routes the transaction to the destination node. Thus, a TLM transaction is implemented to hold the needed information for the routing algorithm such as source, destination addresses and the message contents. The user uses this TLM transaction to implement the messages that move to the upper and lower layers. Free running sequences are used to deliver the messages from a layer to another without changing the message contents.

By applying the proposed solution on layered protocol model, each block inside the layer will be able to communicate with the peer or adjacent layer blocks. The communication will be done through a built-in subroutine call like: send_msg(msg_name, destination_node, contents). Similarly, each block will be able to receive messages from the peer or adjacent layer blocks. The received messages are stored inside the component's TLM FIFO and could be easily parsed and handled.

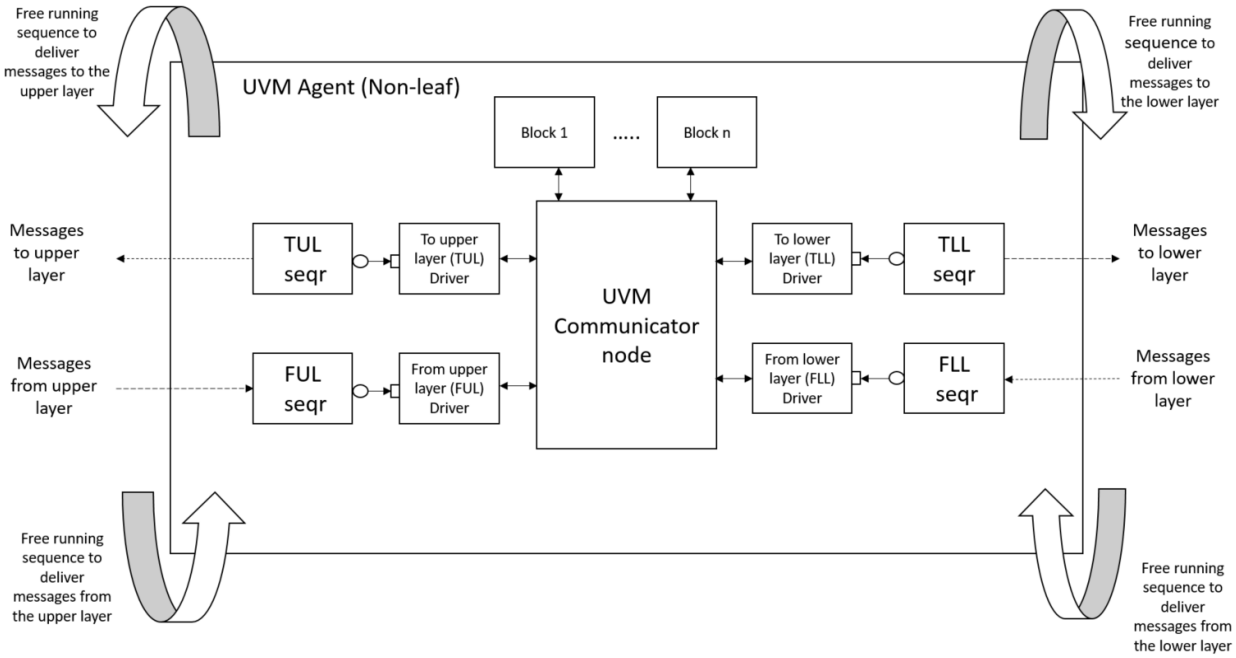

Figure 3. The proposed architecture for UVM Agent

Fig.4 shows how to model the verification environment by using UVM-FM approach. The implemented layered protocol is similar to the protocol shown in Fig.2. In this example, Block A is responsible for managing how to drive the bus in the transmission path (traditional uvm_driver function) , while Block B is responsible for monitoring the bus in the reception path (traditional uvm_monitor function).

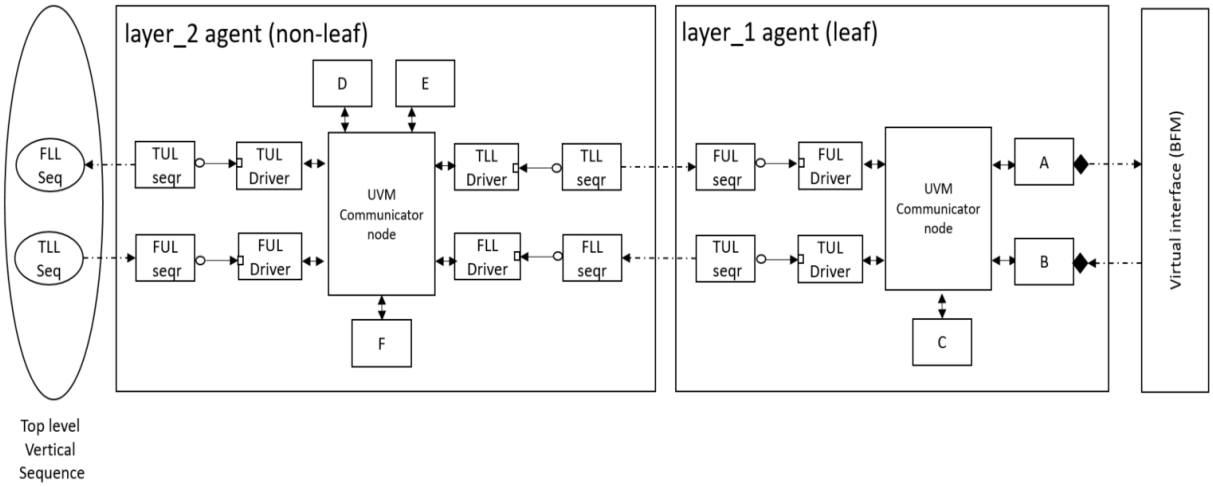

Figure 4. Layered protocol example by using UVM-FM

As shown in Fig.4 the application layer requests are modeled by using the virtual sequence. The top-level sequence could instruct the DUT by sending messages to layer_2 agent "From upper Layer (FUL)" driver. The message received is consumed inside layer_2, and it might trigger other layer nodes to send messages to layer_1 agent through TLL driver. Similarly, the message received to layer_1 agent FUL driver might trigger node A to drive the bus.

On the opposite direction, Block B monitors the bus and constructs messages, these messages could be sent to peer nodes like node A or C, or could be forwarded to layer_2 by using "To upper Layer (TUL)" driver. When layer_2 agent FLL driver receives a message, it forwards the message to one of the layer nodes like D, E or F, and then one of these nodes could send a notification message to the application layer by using TUL driver.

It is observable that many components in the above approach could be reused such as the communicator node, TUL, FUL, TLL, and FLL. Those new elements could be bundled with original UVM package to create a new reusable package (UVM-FM). If such bundle exists, the user will focus only on implementing the system nodes such as A, B, C, D, E and F nodes in the above example. Section V provides the specifications of the new package.

The suggested solution could be used to model more complex layered structure, such as a tree structure where the stacked protocol includes peer layers. For example, layer_2 agent in Fig.4 could communicate with multiple instances of layer_1 agent. Another example for the complex layered structure when the stacked protocol has a management layer that communicates with all the stack layers. For example, a management layer that controls both layer_1 and layer_2 in Fig.4. Serial Attached SCSI (SAS) [2] is an example of a real industry protocol that has a stack with tree structure and includes a management layer as well. The network-like topology for UVM-FM agent gives us the luxury to connect the layer with multiple layers and simplifies modeling any layered structure.

V. UVM-FM PACKAGE

The UVM for functional modeling package (UVM-FM) is a reusable extension layer for UVM. It simplifies the functional modeling by providing a ready-made communication system between the different nodes. The package contains the implementation for the UVM Communicator component (fm_communicator), routing algorithm, logging mechanism, standard system message (fm_message). The package also contains extended versions of the UVM Component (fm_component) and the UVM Driver (fm_driver). Fig.5 shows the UVM-FM Package contents.

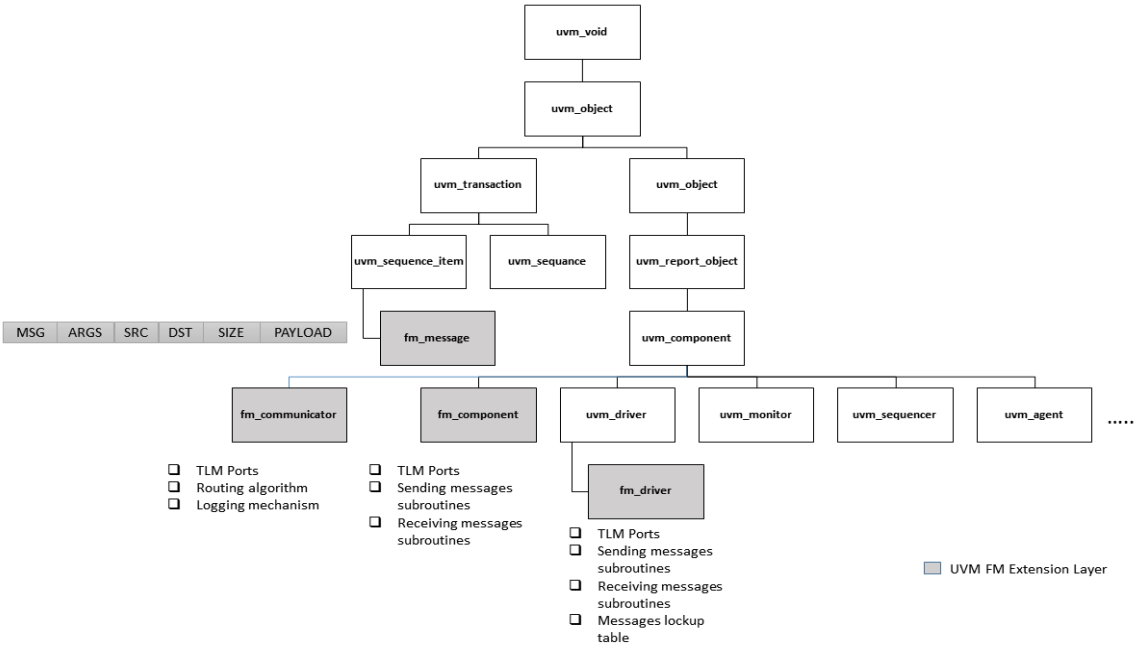

Figure 5. UVM-FM Package Contents

A. Nodes and Messages

A Node is an attached component to the layer communicator, the node could represent a certain internal function for the layer, or it could be used like a traditional driver or monitor in the typical UVM agent architecture.

A message is an information unit to another node that could hold notification about an event occurred, request to perform a certain task, or a confirmation that a previously requested task is done. The user could design the messaging system by using names holding the full information such as TRANSMIT_SHORT_FRAME or TRANSMIT_LONG_FRAME. Another approach could be used by dividing the information into two parts such as a message with name TRANSMIT_FRAME and arguments either, SHORT or LONG.

The user needs to define the system messages and nodes based on analyzing the protocol specifications and extracting the communication messages between the different layers. FM_NODE is a user defined enumerated type that defines the system nodes, while FM_MSG_TYPE is a user defined enumerated type that defines the system messages.

B. Standard TLM transaction (fm_message)

The standard TLM transaction (fm_message) extends the UVM sequence item class. It defines generic fields for the message. UVM_FM package defines the fm_message class and the utility functions such as do copy and convert to string. Table (1) shows the fm_message contents.

TABLE I
FM_MESSAGE CONTENTS

Field Name	Data Type	Description
from_node	FM_NODE	The source node
to_node	FM_NODE [$]	Array of nodes represents the distention nodes
m_type	FM_MSG_TYPE	The information that should propagate between the source and distention nodes
payload	Dynamic array of bits	The message payload
args	bit [9:0]	The message arguments
payload_size	int	The payload size in bits

The "payload" could contain the data received, the data required to be transmitted, or another user data structure (sequence item). During the transmission, the user needs to pack the user-defined sequence item and put it on the "payload" field. In the reception process, the "payload" data could be unpacked to the user-defined sequence item. The "args" field could be used as a whole to define 1024 arguments for the message, or each bit could individually represents an argument to define ten concurrent arguments per each message. The user controls how to parse the message contents such as "payload" and "args", the parsing mechanism will be part of the node definition.

C. The communicator node (fm_communicator)

The UVM_FM Communicator (fm_communicator) is extended from uvm_components to act like a UVM router. Unlimited number of nodes could be connected to this communicator. When a new communicator node is created, the user should register the system nodes by calling a function register_node() which links the node name to a pair of the communicator TLM ports , this pair handles the communication between the node and the communicator.. The communicator receives a message from a node, reads the "to_node" message field, and re-directs the message to one or more receivers. Fig.6 shows the data flow when node A requests to send message M1 to node B, and node B requests to send message M2 to node A and C through the communicator.

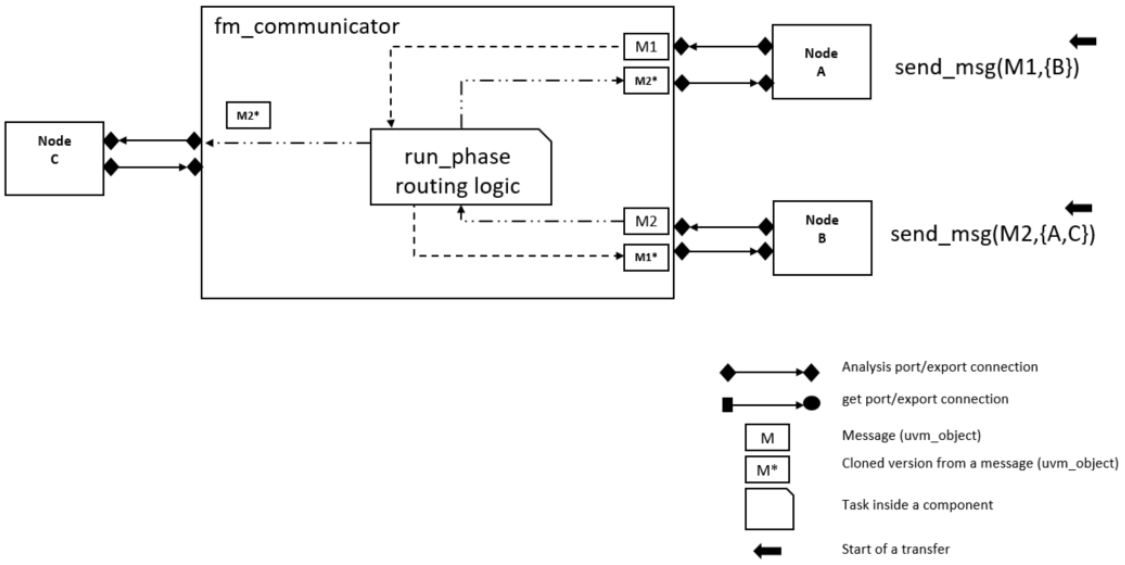

Figure 6. Messages routing in fm_communicator

To facilitate creating new communicator nodes, "fm_macros.svh" macros file should be included in UVM-FM package that implements simple macros. This file contains macros for creating, registering, and connecting ports based on the communicator implementation. Fig.7 shows the contents of a new user defined communicator node

that controls the communication in a system containing three nodes A, B, and C in addition to FUL and TUL driver nodes.

```
class user_mc extends fm_communicator;
      `uvm_component_utils(user_mc)

      //create ports to nodes A,B,C and drivers FUL and TUL
      `create_node_ports(A)
      `create_node_ports(B)
      `create_node_ports(C)
      `create_node_ports(FUL)
      `create_node_ports(TUL)


    function new(string name = "", uvm_component parent = null);
      super.new(name,parent)  ;
    endfunction

    function void end_of_elaboration();
         //register nodes A,B,C and drivers FUL and TUL
         `register_node(A)
         `register_node(B)
         `register_node(C)
         `register_node(FUL)
         `register_node(TUL)
    endfunction

endclass
```

Figure 7. Creating new communicator node

When the user creates an object from the user-defined communicator node inside an agent, the user should use another macro in the connection phase to connect the nodes to the communicator. Fig.8 shows how the communicator is connected to agent nodes.

```
  //define the system components
  user_mc                 comm     ;
  user_compA              A        ;
  user_compB              B        ;
  user_compC              C        ;
  my_ful_driver           FUL      ;
  my_sqr                  sqr_ful  ;
  my_tul_driver           TUL      ;
  my_sqr                  sqr_tul  ;



function void connect_phase(uvm_phase phase);
    //connect the system components
    `connect_comm_to_node(comm,A)
    `connect_comm_to_node(comm,B)
    `connect_comm_to_node(comm,C)
    `connect_comm_to_node(comm,FUL)
    `connect_comm_to_node(comm,TUL)
    FUL.seq_item_port.connect(sqr_ful.seq_item_export);
    TUL.seq_item_port.connect(sqr_tul.seq_item_export);
endfunction
```

Figure 8. The connect phase for an agent built by using UVM-FM package

Since the communicator node is a centralized component for all agent communications, it could log a debugging file that contains the different transfers. That feature eases debugging the model by exploring the different internal transfers and suspecting where the model misbehaved. Fig.9 shows an example for a communicator log file that contains the necessary information about the different propagated messages.

```
================================================================================
MSG routed @time=200

----      COMMUNICATOR MSG -----
-         FROM : A
-         TO   : TUL
-         MSG  : M3
-         ARGS : 0000000000
-         SIZE : 0
-----------End MSG -------------

================================================================================
MSG routed @time=300

----      COMMUNICATOR MSG -----
-         FROM : FUL
-         TO   : A
-         MSG  : M3
-         ARGS : 0000000000
-         SIZE : 0
-----------End MSG -------------

================================================================================
```

Figure 9. Example for the log file generated from the communicator node

D. The component node (fm_component)

The UVM_FM component node (fm_component) is extended from uvm_component; this node could be attached to a communicator node. The user will control parsing the incoming messages by overriding the virtual function msg_decoder(), and could send messages to the communicator node by calling function send_msg(). As shown in Fig.4, nodes A, B and C are connected to the communicator node by using two TLM analysis ports. By using `connect_comm_to_node macro a user could easily attach the component to the agent network. Fig.10 shows an example for a user-defined node B that is receiving messages M1 and M2, while sending message M3.

The node could be used to act like a traditional uvm_driver in terms of converting TLM transactions to signal-level, or to act like a traditional uvm_monitor to convert from signal-level to TLM transactions. For example, in Fig.4, node A acts like a driver and node B acts like monitor.

The node could be used to model an internal function for this layer, instead of modeling the layer in a single component, the user can divide the layer into sub nodes and manage the communication between these nodes by using the communicator. Dividing the layer into sub nodes will ease the development, debugging and allows using the factory override UVM feature [1] to change the behavior of the model by overriding the type of a certain node.

```
class user_compB extends fm_component ;
      `uvm_component_utils(user_compB)

    event e_M1_is_here ;
    event e_M2_is_here ;

    function new(string name = "", uvm_component parent = null);
      super.new(name,parent)  ;
    endfunction

    // msg_decoder    : - user should overide this function to inform the node
    ///how to handle the incoming messges
    virtual task msg_decoder( fm_msg m) ;
       case (m.m_type)

            M1 : ->e_M1_is_here;
            M2 : ->e_M2_is_here;
            default : begin
                        $error("[Comp B] : unexpected msg=%s",m.m_type.name);
                      end

       endcase
    endtask

    virtual task run_phase (uvm_phase phase) ;
           super.run_phase(phase);

           forever begin
             fork begin
               fork

                   begin
                     @e_M1_is_here ;
                    $display("[Comp B] : receving M1");
                   end
                   begin
                     @e_M2_is_here ;
                    $display("[Comp B] : receving M2");
                   end

               join_any
               //sends a message to the UPPER layer (TUL)
               m = fm_msg::type_id::create("m") ;
               m.to_node      = {TUL} ;
               m.from_node    = this_node;
               m.m_type       = M3 ;
               m.payload_size = 0 ;
               send_msg(m) ;
               disable fork ;
             end join
            end


    endtask

endclass
```

Figure 10. Example for a user defined node

E. The communicator driver (fm_driver)

The traditional uvm_driver is responsible to grab a sequence item from the attached sequencer and translate it to signals level. In UVM-FM, the TLM to signals level translation is delegated to another component node while the fm_driver is responsible to manage the communication between the different system layers. As shown in Fig.3, in non-leaf agent there are four types of driver from upper layer (FUL), to upper layer (TUL), from lower layer (FLL) and to lower layer (TLL) drivers. Both TUL and TLL are forwarding the received messages from the communicator node to upper and lower layers respectively, while both FUL and FLL are receiving messages from the upper layer

and lower layer respectively. Both FUL and FLL decide which nodes in the agent would be interested in this message then sends the message to the communicator with a modified list of reception nodes.

Fig.11 shows data flows for two messages, node A sends a message M1 to the upper layer (TUL), while node B receives message M2 from the upper layer (FUL). Nodes A and B aren't aware about the internal structure for the upper layer, thus they deal with TUL and FUL drivers which are connected to the upper layer FLL and TLL drivers respectively, the connections are done through free running sequences as mentioned in section IV. When node A decides to send message M1 to the upper layer, the communicator node forwards this message to the TUL driver, which is blocked until receiving a message from the communicator, the TUL finishes the opened sequence item on it and allows the free running sequence to propagate the information to the upper layer. When FUL driver receives a message from the upper layer, it searches on a user pre-defined lookup table to decide the recipient nodes for the received message; this lookup table represents the interface specifications between the two layers. The FULL driver overrides the message contents by modifying the "to_node" field with the results found in the lookup table , then it sends the message to the communicator.

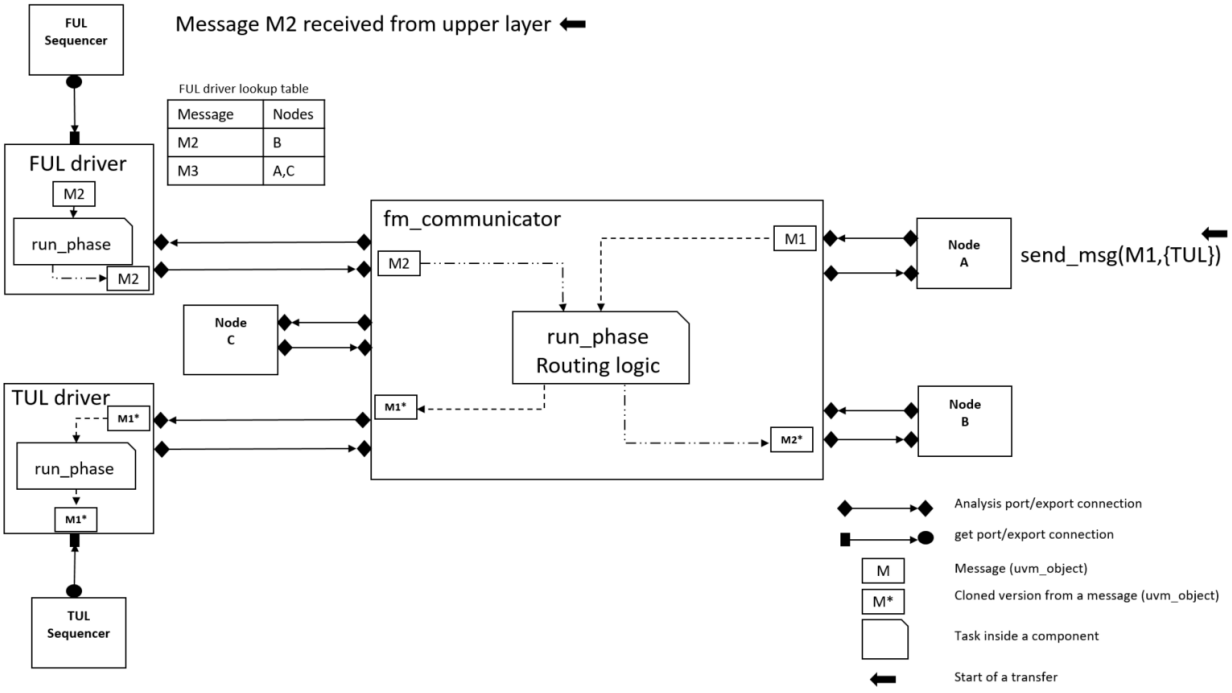

Figure 11. Connecting FUL and TUL drivers to the agent's network

Fig.12 shows two user defined drivers by using UVM-FM. In my_full_driver implementation, register_msg() function is used to build the lookup table for the FUL driver , while send_msg() is used to forward the message to the communicator. The implementation of fm_driver::send_msg() differs than fm_component::send_msg(), in the fm_driver method the "to_node" is overridden based on the FUL lookup table. In my_tull_driver implementation, the task got_msg() is used to block the driver until a message received from the communicator.

```
class my_ful_driver extends fm_driver #(fm_msg) ;

    `uvm_component_utils(my_ful_driver)

    // Standard UVM Methods:
    function new(string name="my_ful_driver", uvm_component parent = null);
      super.new(name, parent);
    endfunction

    task run_phase (uvm_phase phase) ;
        super.run_phase(phase);
        //register msg M3 to be sent to nodes A and B
        register_msg(M3,'{A,B});
        forever begin
            seq_item_port.get_next_item(req);
            //sends the req to the communicator to get handled by this layer
            send_msg(req);
            seq_item_port.item_done();
        end
    endtask

endclass


class my_tul_driver extends fm_driver #(fm_msg) ;

    `uvm_component_utils(my_tul_driver)

    // Standard UVM Methods:
    function new(string name="my_tul_driver", uvm_component parent = null);
      super.new(name, parent);
    endfunction

    task run_phase (uvm_phase phase) ;
        super.run_phase(phase);
        forever begin
            seq_item_port.get_next_item(req);
            //wait to get a message from the communicator
            got_msg(req);
            seq_item_port.item_done();
        end
    endtask

endclass
```

Figure 12. Example for user-defined TUL an FUL drivers

VI. CASE STUDY

The proposed solution is used to model verification environment for the link layer of SAS protocol [2]. The link layer consists of ten different blocks. Fig.13 shows the link layer agent in SAS verification environment, which uses the UVM-FM approach.

The UVM-FM package simplified the test environment creation in the following aspects: providing a standard architecture and communication system, simplifying adding and implementing new component to the system, simplifying how to debug the system by adding temporally test nodes to mimic the upper layer, and using the log file generated by the communicator node to debug the environment.

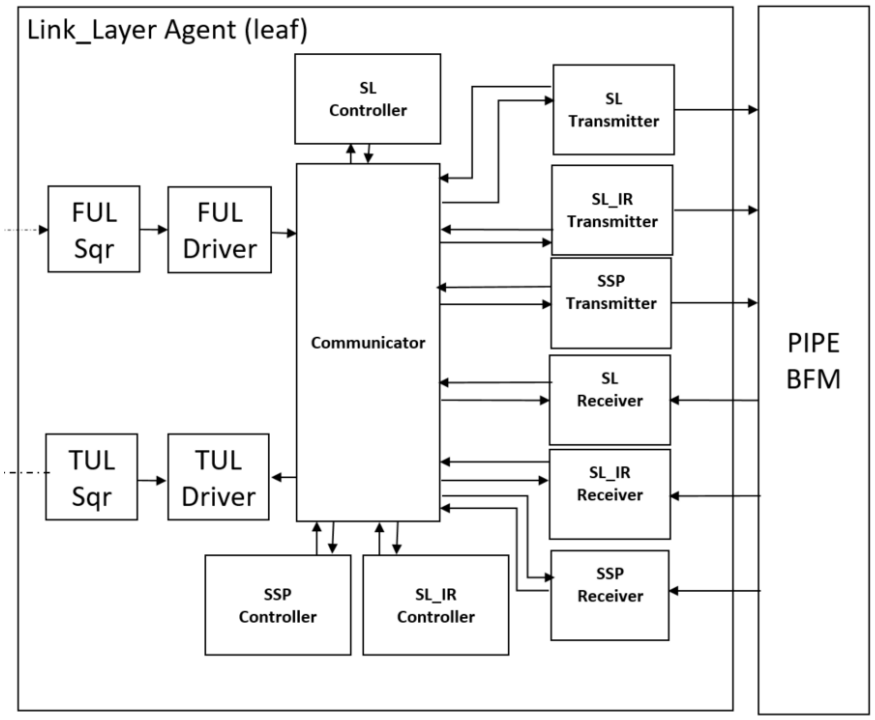

Figure 13. Link Layer Agent by using the proposed modeling approach

Modeling a complex layer like SAS link layer with a single driver would be complex task; each node in the system has an independent state and has the ability to send messages to a peer node or to the upper layer. Without UVM-FM standard communication mechanism, synchronizing the different nodes together will be complex and time-consuming task. In addition to that, it was easy to integrate this layer with the port layer (the upper layer), which has similar network topology and communication system.

VII. FUTURE WORK

This kind of standardization is very useful for complex systems modeling. It is also a step forward for more automation in building testbenches, as it allows building the testbench automatically by giving textual description to the different communications between the system nodes. For the system described in Fig.2, and after well understanding for the protocol, we could extract a textual description for the communication messages between the several IP blocks. Fig.14 shows an example for the needed specifications.

Layers	L1,L2			
leaf layer	L1			
Message	**Arguments**	**Payload**	**From Node**	**To Nodes**
M1	AR1_1,AR1_2	yes	L1.A	L2.E
M2	None	None	L.1.B	L1.C
M3	AR3_1	None	L2.D	L2.F
M4	None	Yes	L2.F	L2.E,L1.A

Figure 14. Example for the example IP extracted specifications

From the extracted specifications, we can know that the system contains two layers L1 and L2; L1 is a leaf layer. In addition to that, we can know that L1 contains blocks A, B and C while L2 contains nodes D, E and F. We can also know the received and transmitted message for each node, we can expect that L1 FUL driver needs look-up table to re-direct message M4 to block A, and L2 FLL driver needs lookup table to re-direct message M1 to Block E. The extracted specifications contain all the needed information to construct the environment showed in Fig.4, this process does not need a human effort, a script could be written to convert this textual description to a UVM-FM based skeleton. By using this automation approach, the user will focus only on implementing the nodes while the other system components will be auto-generated.

UVM-FM is extendable approach; it could be used for protocols with more than two layers. Theoretically, it could be used for protocols with a management layer that accesses all the protocol layers or with protocols with tree structure. These kind of complex topologies should be examined in the future work. Fig.15 shows an example for the management layer.

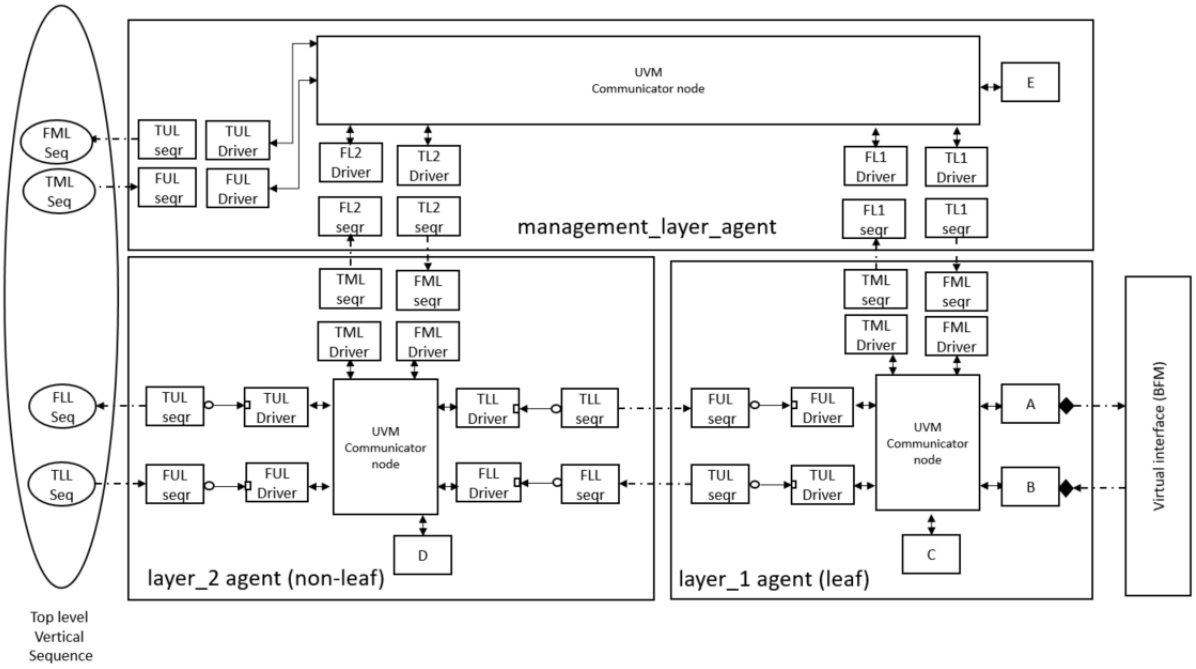

Figure 15. Example for complex protocol with management layer

VIII. CONCLUSION

UVM-FM is an approach to facilitate modeling layered protocols by using UVM; it solves the complexity of managing bi-directional communication channels between the different protocol agents. This paper introduces the related work for modeling layered protocols and shows how UVM-FM contributes to give a standard solution for modeling such protocols. The paper provides specifications for the UVM-FM package, which is a reusable extension layer UVM that contains new suggested components such as "fm_communicator", "fm_component" and "fm_driver". UVM-FM has been used on real-word industry protocols and eased implementing and debugging complex systems.

REFERENCES

[1] IEEE 1800.2 Standard for Universal Verification Methodology, February 2017
[2] SAS Protocol Layer - 4 (SPL-4), Revision 09, July 2016
[3] Tom Fitzpatric, "Layering in UVM", Verification Horizons
[4] Doulous, "Requests, Responses, Layered Protocols and Layered Agents", Online resources at http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/layering
[5] H.Yu and C.Thomson, "A Simplified Approach Using UVM Sequence Items for Layering Protocol Verification," DVCon USA, February 2017
[6] Rahul Chauhan, Grupreet Kaire, Ravindra Ganti, Subhranil Deb, "Layering Protocol verification: A Pragmatic Approach Using UVM", SNUG 2014