Customizing UVM agent to support multi-layered & TDM protocols

Amit Pessach, Veriest Solutions, Israel, amitp@VeriestS.com

Danilo Cirovic, Veriest Solutions, Serbia, danilo@Veriests.com

***Abstract*— This paper addresses implementation issues faced when dealing with layered agent architecture and channelized agent definition. Advanced methodologies will be demonstrated using advanced UVM capabilities, which most basic agent’s implementation in the industry don't commonly used.**

Keywords—SystemVerilog, UVM, layer

I. Introduction

Today’s functional verification became very complex as the designs are getting more and more complicated. As result, verification languages and associated methodologies are improving for the verification engineers to be able and implement more modular, reusable, efficient and high-quality verification environments.

As result from the high complexity of the verification requirements, standardizing verification language and methodology became essential.

It is very hard to standardize a full verification language and methodology which covers all aspects and requirements of any verification environment.

System Verilog verification language was standardized in order to become an industry common standard for verification.

On top of System Verilog UVM was standardized as well as the common leading verification methodology for verification environments.

Due to the complexity of the defined methodology some definition holes still exists, which gives some freedom for the verification engineer on one hand but causing some differentiations between different implementations on the other hand.
Due to complexity of defining a good and simple user interface for methodology usage, a lot of “behind the scenes” implementation was required to be implemented as part of UVM infra structure.
For common usage UVM infra-structure API is mostly sufficient but when getting into some more complex and corner case usage models, sometimes the user interface is not sufficient and user need to investigate on how to use the methodology infra structure in more sophisticated way.

II. Problem Statement

UVM methodology defines a modern architecture for implementation of an architectural agent.
In general an agent is responsible for handling all related aspects of a pre-defined physical interface.
Basic architecture defines couple of main elements as part of a classical agent:

- A dedicated data item (sequence item) for transaction modeling of the data transferred over the bus.
- A dedicated System Verilog interface containing all required signals, clocking blocks, mod ports and other physical related support such as low-level protocol assertions, physical protocol related checks and timing requirements.
- A driver handling all physical layer aspects, bus functional modeling and full protocol specific handshake support.
- A monitor handling transaction's collection from the bus connected to an analysis port for collected data spreading.
  In some cases monitor also performs collected data integrity checks.
- A sequencer for sequence execution and sequence's arbitration support.
- Sequence library containing basic sequences with transaction creation usage model examples.
- A configuration object for controlling agent's operation and providing required controllability for the user.

Below are main concepts any agent which is modeling an interface should consider and support:

- Handling full interaction with the predefined System Verilog interface.
- Support both initiator/master and responder/slave functionality as the protocol defines.
- Must not be DUT or specific design biased.
  A good agent should be compatible with any design which includes the interface the agent is supporting and should not be tightly coupled with a specific design (for example, specific design configuration which are not part of the protocol should not be handled by the agent layer).
- Implementation and usage should be as much as simple but still supporting full interface and protocol view.
- Providing full protocol checks and transaction validity related checks.
- Should be easy to integrate in all levels (i.e support easy and quick vertical reuse).
- Should be reusable (i.e support easy and quick horizontal reuse).

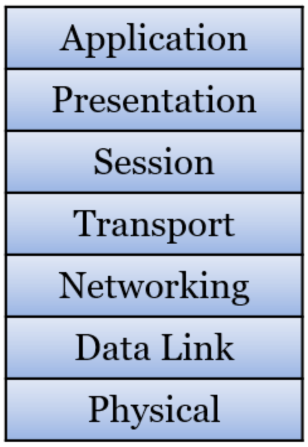
With the above in mind, UVM provides required support for any agent implementation but there are still some special agents which requires functionality that basic UVM support might not be enough.

This paper shares such implementation dilemmas and solutions for cases which classic agent architecture does not provide the straight forward required support.
These implementation cases require deeper understanding of UVM infra structure in order to find the most simple, generic and reusable solution.

III. Multi Layered Agent Architecture

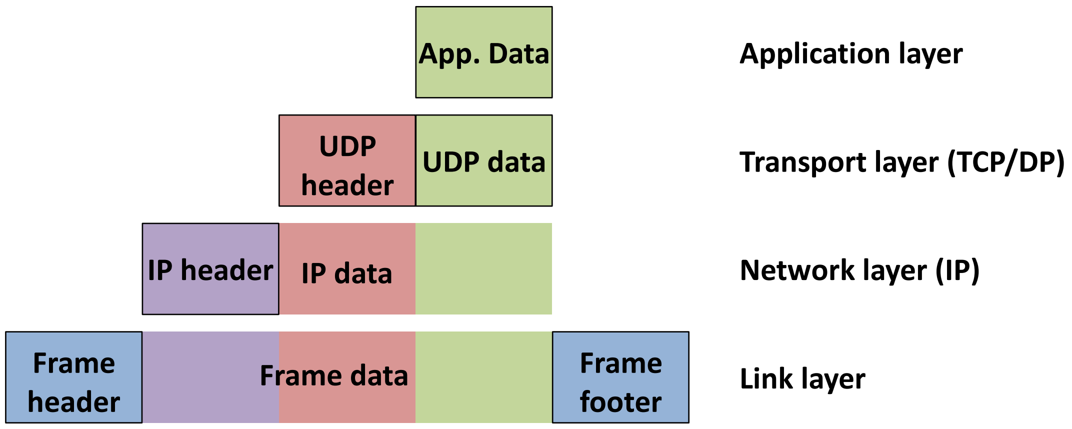
In the DataCom world most of the common used network protocols enable layering.
Multilayer network communication protocols provide standards that allow diverse systems to communicate with each other.
Each layer is responsible for handling encapsulated data relevant to its functionality only.

Protocol layering is defined by the OSI seven layers model shown below:



Ethernet is the most popular layered protocol commonly used all over the industry.

Ethernet layering is demonstrated below:



IV. MULTI-LAYER - "COMMON VERIFICATION APPROACH"

The "common" and obvious approach is implementing functionality of all layers in a single agent.

This implementation is the simple one from architecture point of view and UVM infra structure usage, but implementation might get too complex and might have many disadvantages such as:

- Facing difficulties to control each layer's functionality especially lower layer behavior.
  Lower layer functionality and associated data structures will have limited control.
- Often agent's usage might be focused on lower-layer testing only which might be not applicable.
- Relations between data structures of different layers become very limited.

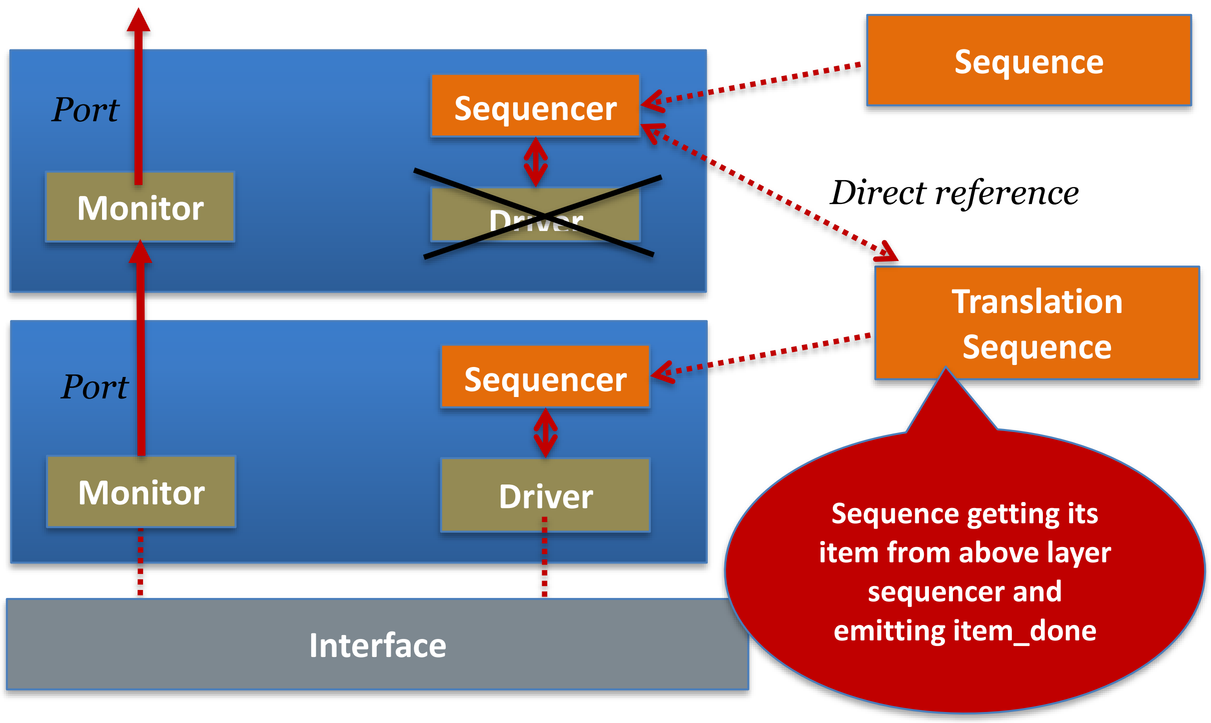
I this way of implementation, each of the agent's components previously described will support functionality of all layers (i.e single data item and configuration object include fields and member of all layers, driver and monitor supporting functionality of all layers, etc…).
Above makes each component very complex, hard to maintain and control.

This kind of approach might be enough for many agents but in some cases is not flexible enough to achieve high quality verification capabilities.

V. MULTI-LAYERED RECOMMENDED ARCHITECTURE

Before starting multi layered implementation first need to answer the question of whether a multi-layered agent architecture should be used for the specific case.
Need to consider main pros and cons of this kind of implementation.
The difficulty is mainly about finding the right balance between abstraction and fine grain control of the stimuli creation.
When using a single agent for a multi layered application, users might find it very difficult to control lower layer behavior from high layer data structures, hard to control specific layer attributes and hard concentrate on single layer testing.
These and some more reasons are why this approach might not be flexible enough to achieve high quality verification abilities.
On the other hand, in cases in which layered architecture is not required, implementing it might be an unnecessary over-design.
For adopting layered UVM architecture in which each layer has its own agent capabilities, there are some issues that need to be considered which UVM classic agent functionality does not offer.
Translation sequences should be used between the layers, as well as deactivation agent's driver for preventing multiple sources of transactions requests from the upper layer sequencer.

In case decided on this approach will come up with the following architecture:

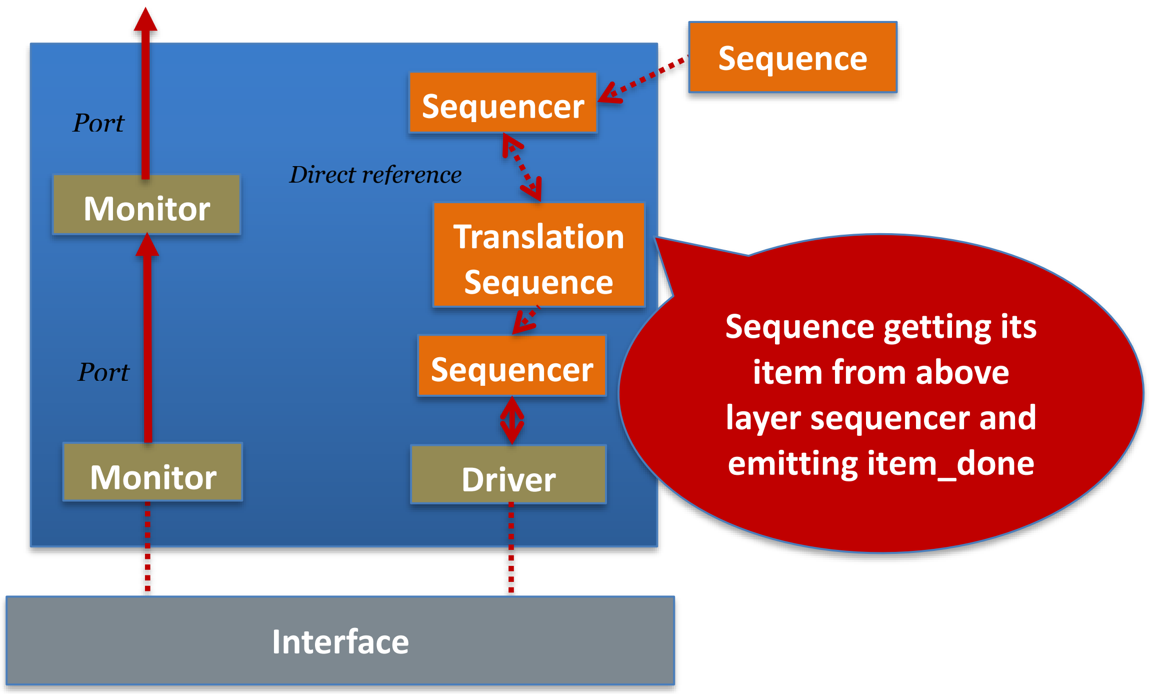


Main implementation principles of the above architecture are listed below:

- Each layer modeled by a different agent.
- Different transaction type (sequence item) for each layer.
- Each layer has its own sequence for transaction generation and stimuli control.
- Each agent is unaware of implementation details and functionality of its neighboring agents.
- Interaction between layers based on pre-defined API decided up front required for data transport and data conversion between layers.
- Lowest layer agent maintains active components as any “standard” agent as it handles a physical interface.
  Higher layers activate its generation components (sequences layer and sequencer) excluding driver.
  Higher layers provide any required support for implementing a conversion sequence for the layer beneath.
  The translation sequence is getting data object of its upper layer and transforms it to its own data object by applying all required manipulations.
  Monitor of all higher layers is not connected to a physical interface as usual but to an analysis port and should be able to process it, process relevant data and pass only encapsulated data to the monitor of the layer above.

There are some cases in which network higher layer protocols can be transported on different lower layer protocols but in some cases layers are statically defined.
For example, PCIe Transaction Layer Packet is intended to be transported by PCIe Data Link Layer only.

In such cases there is no justification for maintaining separate agent for each layer and layering is done within a single agent.
This result with the following architecture:

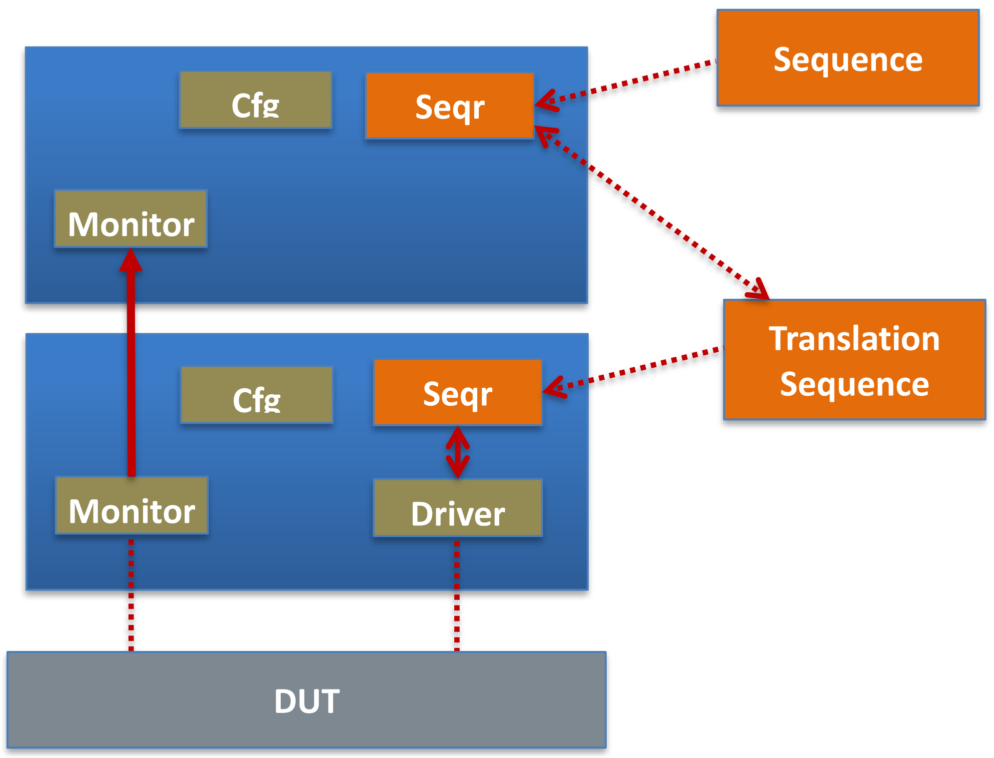


Note that all implementation principles are like the case of an agent per layer.

VI. MULTI LAYER AGENT PRACTICAL USAGE AND DEBUG CAPABILITY

Multi-layer implementation enables high quality usage model mainly in means of controllability and debug capabilities which are very hard in other implementations.

Below demonstrated two layers agent connected to a DUT.



Couple of the main advantages are listed below:

- Controlling each data item members separately easily enables controlling behavior of each layer's sequence separately.
- Tune control members in each configuration object separately easily enables controlling each layer full behavior without any dependencies of other layer.
- Easily enabling/disabling checks/assertions of each monitor of each layer.
- Controlling debug messages verbosity separately for each layer.
  Very important in case debug is focused on specific layer functionality.

VII. Channelized Agent Architecture

Another area in which basic agent architecture might not be enough is in case of handling channelized protocols.

There are many examples for such protocols.
In the simple case there might be different logical traffic elements over same physical interface such as in band priority mechanism or any other data differentiation.
In more complex cases the physical protocol defines TDM (time domain multiplexing) behavior between logical channels on a shared bus.

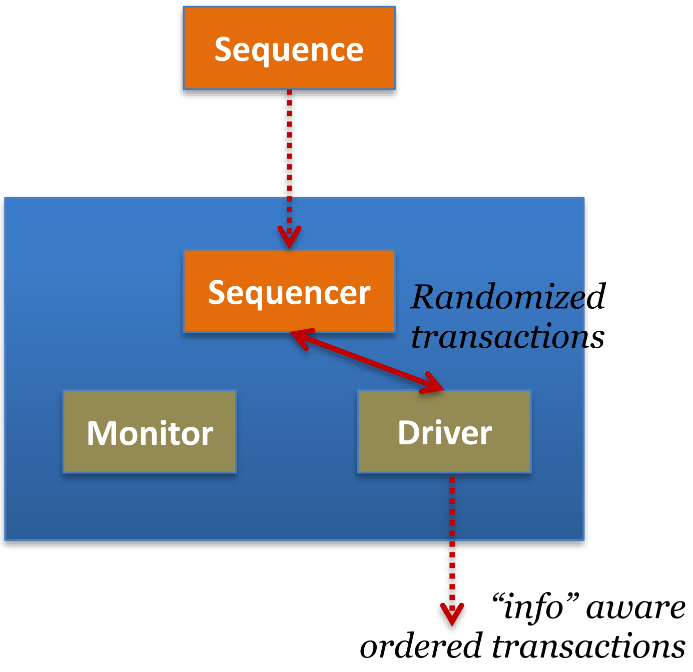
In these cases, using the basic agent architecture might not be the right choice mainly due to the effort required, and might prove insufficient in keeping a clean and modular implementation as well as enabling high controllability for the user.

Problem becomes much more critical when using an existing agent to which above functionality awareness should be added and changing existing agent source code is not an option (i.e. due to encryption, reuse or any other reason).

For example, let's assume on top of an existing packet interface agent required to add some strict packet priority mechanism or define traffic flows each also might be associated with a unique channel/port that has its own time slot on the physical interface.

One possible approach might be defining a layered architecture as demonstrated previously but in some cases this kind of a solution might be too complex and involve unnecessary overhead.

The "common" and straight forward approach is to adjust the agent driver and add all required functionality inside.
This solution requires adjustments within the transaction data type and the driver to support the new feature, as shown below:



Complexity in this approach is implementing all functionality inside the lower level component of the agent (driver) which is usually not the best choice cause dealing with the problem in higher layer is much better from complexity point of view and controllability.

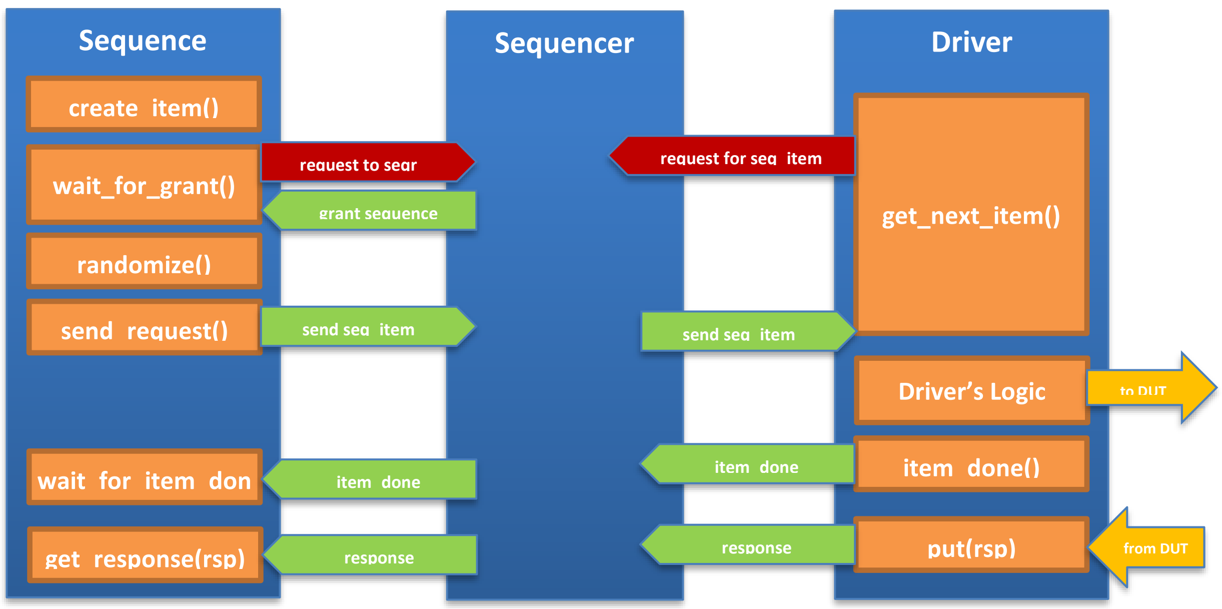
This solution result in very complex, not modular and non-controllable implementation.

On top of that, in case agent is encrypted it is practically not possible to adjust its driver.

More clean, controllable and easy implementation should make use of the internal sequencer arbitration capabilities.

To deeply understand the suggested solution will start with brief description of how driver-sequencer-sequence is executing and handling sequence items.

The handshake between the agent's components is shown in the following figure:



The above illustrates all steps from the start point once a sequence is ready to generate a sequence item till it can move on and generate the next one.
Note that in the figure only one sequence is shown as running on the sequencer.
In real life multiple sequences can be executed in parallel on a single sequencer.
In such a case couple of sequences can request to send an item simultaneously and the sequencer should pick only one at a time and give it a grant while all other will be blocked.
In such a case an arbitration mechanism is triggered within the sequencer between all available sequences.

For the sequence arbitration UVM supports couple of different Sequence Arbitration mechanisms shown below:

- UVM_SEQ_ARB_FIFO
- UVM_SEQ_ARB_WEIGHTED
- UVM_SEQ_ARB_RANDOM
- UVM_SEQ_ARB_STRICT_RANDOM
- UVM_SEQ_ARB_STRICT_FIFO
- UVM_SEQ_ARB_USER

Note that in the default case sequencer is using the first one mentioned which is actually first sequence requesting to send an item will get the grant.
In case couple of sequences are requesting simultaneously there is no guarantee for the execution order.

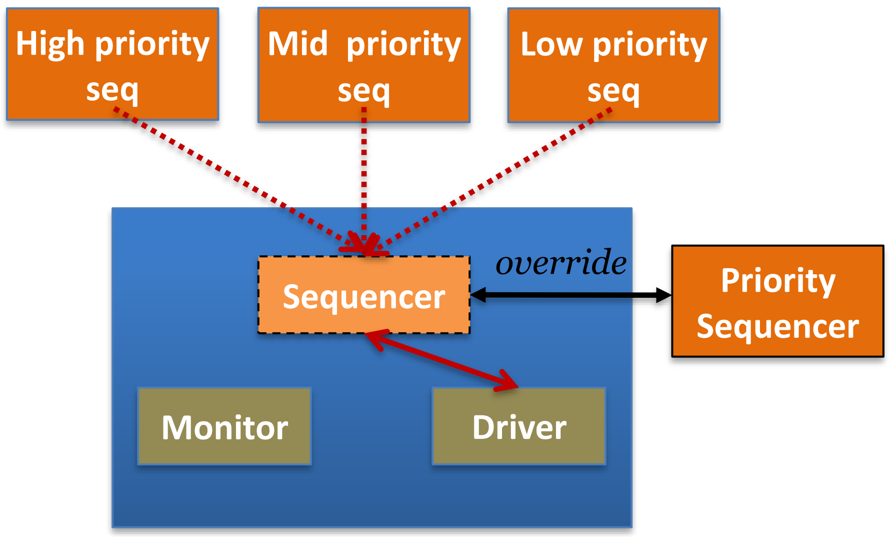
Now we can start looking into cases in which will make use of the above theory.

VIII. Traffic Flows Modeling

Assume requirement is adjusting a simple agent architecture to support multiple traffic flows with user-defined priority between them.

Suggested implementation is creating separate sequences running in parallel while each is handling items from different traffic priority level.
Required functionality achieved by changing default arbitration mechanism within the sequencer by selecting SEQ_ARB_USER mode and overriding its default arbitration behavior.

All modification done in sequence level.
Existing agent untouched and unaware of the added priority functionality as shown below:



The adjustments required keep the agent untouched and are only in the environment level as described below:

- Inherit agent sequencer and change its arbitration mechanism.
  This is done first by executing its ***set_arbitration()*** function with UVM_SEQ_ARB_USER parameter as an argument.

  Once changing the arbitration mechanism as above, a dedicated function of the sequencer will be called automatically each time the sequencer should give grant to one of the sequences executed on it.
  The following virtual function will be triggered should be implemented:
  virtual function integer user_priority_arbitration(integer aval_sequences[$])

  The override implementation should return one of the entries from the ***avail_sequences*** queue, which are indexes into an internal sequences queue ***arb_sequence_q***.

  In our required implementation, function will return the sequence with highest priority value.

- Overriding agent sequencer with the new one created.

- Executing sequences with priority awareness as follows:

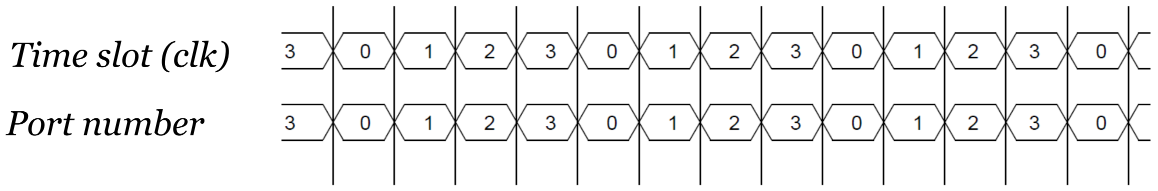
```
high_prio_seq.start(seqr,,HIGH,)

mid_prio_seq.start(seqr,,MED,)

low_prio_seq.start(seqr,,LOW,)
```

  In this way priority awareness is within sequence execution and data item should not be adjusted as well.

IX. TDM MODELING

Assume an existing agent driving packets on four logical ports/channels over a simple interface composed of the following signals:
clk, port_num[2:0], data[255:0], valid
Once a packet started on a specific port it should be completed before starting another packet on same port.

On top of this agent, there is a new requirement of adding TDM behavior.
Ports are sharing the bus using Time Domain Multiplexing technique.
Sharing should be done in round robin mechanism as shown below:



TDM timing/scheduling should continue regardless of whether there are available packets to send.
In case no packets are available for transmission on a single port, a bubble should be created and once packets will be available TDM should continue with same original timing.

As detailed in the previous example, sequencer arbitration functionality should be overridden to execute the TDM scheduling and select each cycle the sequence of the relevant port.

But this implementation is much more complex than the previous one demonstrated because of its corner cases which should be handled carefully.

First is in case there is no available sequences for a specific port?

In UVM, arbitration mechanism **must** grant a pending sequence each time it is triggered.
Implementation should grant sequence only during its TDM slot.
Granting any other available sequence will cause corruption of the original TDM scheduling.
So how should implementation of ***user_priority_arbitration()*** handle this case ?

Solution for that case will be adding "dummy" sequence always available for arbitration creating an item with an invalid port number.
This sequence will be always selected and executed in any case of non-available sequence in order to create a "bubble" on the bus.

Note that for the above solution, driver should be slightly modified to assign idle on the bus in case getting item with invalid port number from the "dummy" sequence.
This adjustment to the driver cannot be eliminated due to the fact there is no other solution to overcome this UVM limitation.

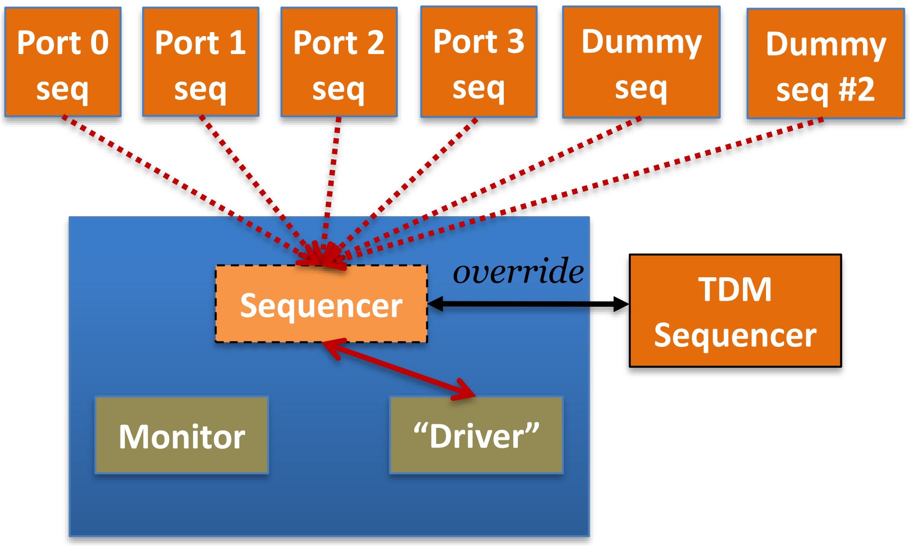
Now what in case there is no available sequences at all?

Only the previous described "dummy" sequence will be available.

In UVM, in such a case of single available sequence, arbitration mechanism will not be triggered at all and as a result will lose track of the TDM scheduling timing.

Solution is to add another "dummy" sequence so always there are at least two sequences running and available. In such case arbitration mechanism will be always triggered and will be able to keep sync of the TDM timing.

Finally ending with the below agent architecture:



X. Summary

This paper focused on implementation issues which we are facing once dealing with layered agent architecture and channelized agent implementation.
Advanced implementation demonstrated using advanced UVM capabilities which are not commonly used in most basic agent's implementations in the industry.

These and some more cases require deep knowledge of UVM infra structure to come up with a clean, simple and reusable implementation solution while using basic UVM usage model might result in much more complex and less quality solution.