Exploring UVM TLM2 based Sequence, Sequencer and Driver in UVM

N. Goyal, J. Refice
Nvidia Corporation
2888 San Tomas Expwy
Santa Clara, CA 95051

***Abstract*- The sequencer and driver communication was adopted in UVM (Universal Verification Methodology) from OVM (Open Verification Methodology). When it was added to OVM, TLM 2.0 (Transaction Level Modeling) did not exist. Hence the communication was based on TLM. When TLM2.0 was introduced to UVM, no attempt was made to 'update' the older TLM based OVM protocols.**

The current protocol between sequencer and driver using the seq_item_(ex)port provides get, put and peek but isn't strictly UVM TLM1. The back-and-forth communication between the sequence and the sequencer requires handshake which adds complexity and isn't handled properly by UVM TLM1.

This paper proposes backward compatible updates to the UVM library designed to incorporate TLM2.0 support into the sequence-sequencer-driver stack, enabling utilization of TLM2.0 features.

I. INTRODUCTION

The existing sequence - sequencer - driver communication is TLM1-*like* at best [1]. The sequence<->driver stack is reasonably well described, but it doesn't map well to TLM and introduces time consumption in places where it isn't strictly needed. The communication between sequencer-driver is handled via a special interface and ports/exports that were inspired by TLM1. We will analyze the current communication between sequence-sequencer which is a bi-directional handshake and complex. We will also explore the possibility of mapping these protocols to UVM TLM2 and the special handling needed.

The motivation to explore the possibility of mapping this communication to UVM TLM2 comes from the fact that TLM 2.0 allows for non-blocking (zero-time) communication and has clear notifications for the beginning and end of a transaction which helps simplify timing semantics [5]. TLM2.0 presents a more robust means of communicating protocol-level information. TLM2.0 communication allows for multiple responses, reduces the need to map a response to its request and would also allow communication with a TLM2 initiator.

Note – When referring to System C, it's TLM and TLM 2.0. When referring to UVM, it is UVM TLM1 and UVM TLM2. In this paper, we will use TLM1 and TLM2 to refer to UVM TLM1 and UVM TLM2, respectively.

II. WHY TLM2?

Our exploration into the feasibility of adopting TLM 2.0 for the standard UVM sequence-sequencer-driver communication is motivated by two key considerations: addressing certain limitations of the current TLM 1.0-like protocol and leveraging the advanced features of TLM 2.0.

Specifically,

i. *Timing Semantics* – The current implementation forces the consumption of time, like consuming delta cycles in methods like try_next_item() to unblock the sequence. Stacking sequencers too can consume delta cycles for synchronization. This can lead to undefined behavior. TLM2 provides non-blocking function calls, with clear beginning and end of transaction phases allowing for a more transparent and deterministic representation of a transaction's life cycle.

ii. *Response Mapping and Multiple Responses* – Each time a response is received, it must be mapped to its respective request using the specific sequence_id. This implicit mapping is prone to errors if not handled correctly and can become complex to debug. TLM2 provides the ability to package the response within the payload, making the process more robust and intuitive. The current API also has no way for the sequence or the sequencer to indicate that the given transaction is complete. TLM2 uses return state to indicate the completion of a transaction, thus making it easier to track and less error prone.

iii. *Interoperability* – Adopting native TLM 2.0 sockets for the core sequence-sequencer-driver path would standardize this communication at the architectural level, potentially streamlining mixed-language verification environments and component reuse.

III. SEQUENCE-SEQUENCER-DRIVER IN UVM

A. Sequencer - Driver Communication

The existing sequencer-driver communication uses a specialized interface for communication but does not use the standard TLM1 protocol. UVM defined its own specialized protocol for driver sequencer handshake, `uvm_seq_if_base#(T1,T2)[3],[4]`. The handshake between the driver and. the sequencer requires bi-directional communication for a synchronized handshake Here is how it works today:

i. `get_next_item, try_next_item, peek, get` - A driver asks for the sequence item when it is ready to drive the transaction on the interface. This method gets unblocked only when there is a sequence item available.

ii. `item_done` - The driver must indicate to the sequence, via the sequencer, that it may now try and send another request item to the driver. This call unblocks the sequence, and sequence can move forward to generate the next item. It should be noted that functionally, `get` call is equivalent to `get_next_item` immediately followed by `item_done`.

iii. `put, put_response, write` - If there is a response received on the interface, the driver has an option of sending it back to the sequence via the sequencer calling `put_response` on the sequencer port or write on rsp_port.

iv. `has_do_available` - A non-blocking method to determine if there are any sequences attempting to send an item. Note that it doesn't guarantee that an attempt to retrieve the next item will succeed in zero time, but if it returns 0 then the attempt will likely consume time.

Noticeable drawbacks of using TLM1 for our driver and sequencer communication are -

- `try_next_item` is a task. The LRM dictates that the sequence can't consume real time between `wait_for_grant` returning and `send_request` being called. However, in the current implementation, `try_next_item` needs to consume delta cycles so that the sequence's body can be unblocked after `wait_for_grant` and then call `send_request`.
- `has_do_available` does not consume time. However, even if `has_do_available` returns 1, time may be consumed before an item becomes available.
- `item_done` does not necessarily imply that the item has been driven on the interface. It simply means that a sequence is now allowed to attempt and send another request item to the driver. This lack of a strict definition causes inconsistent implementations.

B. Sequence - Sequencer Communication

There are various APIs available for sequences, such as `` `uvm_do, `uvm_create/`uvm_send `` and `start_item/finish_item`. Looking at the code, the hierarchical structure of the methods looks like this [3],[4]:

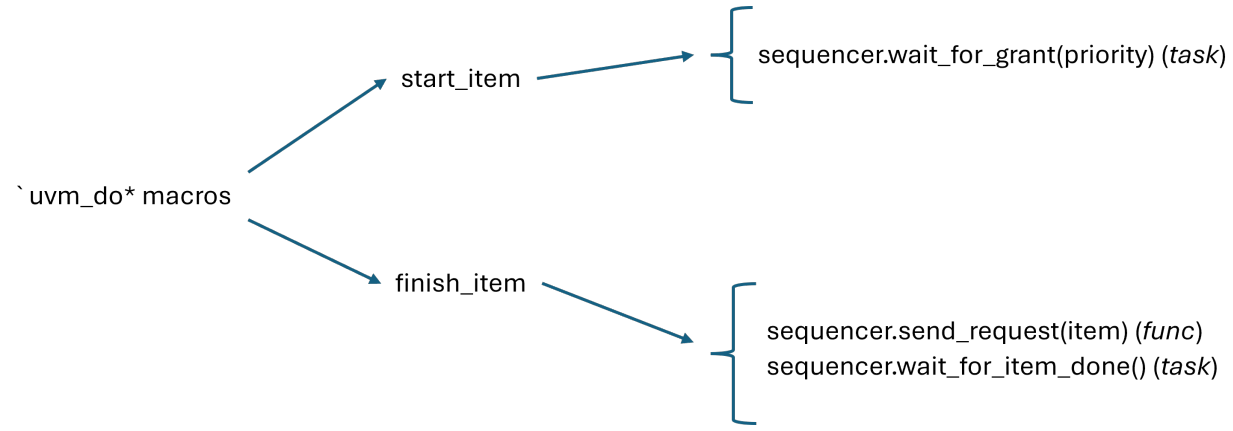

Figure 1: APIs called in the sequence

The final three methods communicating with the sequencer as per Figure 1 are:

i. `wait_for_grant` – This method waits until the sequencer finishes arbitration. It is called to check if the driver is unblocked and ready to send out an item. The item need not be created prior to calling wait_for_grant.

ii. `send_request` - This is a zero-time function call. As soon as `wait_for_grant` returns indicating that the driver is ready to accept a transaction, the sequence prepares an item to be sent out. The sequence waits to prepare an item until this call to allow for late randomization.

iii. `wait_for_item_done` - This task may be called after the request is sent. Once the driver sends `item_done`, this task gets unblocked. Note that it does not imply that the item is sent out on the driver. It just blocks sequence from continuing. If the sequence does not wait for `item_done`, the fundamental protocol stays the same.

C. Putting it all together

The communication between a sequence, sequencer and driver requires more than the uni-directional protocol of TLM1. The current protocol between the sequence/sequencer and sequencer/driver is a customization of TLM1 to accommodate for its special needs. The current implementation is also TLM1-*like* but not TLM1. For instance, there is no `try_get` or `can_get` in this implementation and the method that exists, `try_next_item`, is not equivalent of `try_get`.

In spite of being customized, UVM has two completely different APIS to communicate between a sequence/sequencer and a sequencer/driver. This is unnecessarily complicated and makes it harder for the user to understand the protocol.

The customized TLM1 in UVM addresses:

i. Bi-directional handshake

The interaction between a sequencer and a driver needs to be bi-directional to enable late randomization and provide the ability to send an updated item. Similarly, the interaction between a sequence and sequencer needs bi-directional interaction to enable late randomization and relevance checking for the sequence. The standard TLM1 ports offer one way communication via `put, get, peek` methods. Whereas the methods used today are `get_next_item, try_next_item, item_done` and `put_response` along with the previous existing `get, put` and `peek`. Method `get_next_item` for instance, is called when the driver is ready and unblocks only when the sequence has delivered the next sequence_item. Thus, enabling a bidirectional handshake.

ii. Late randomization

In the current implementation, the `wait_for_grant` doesn't send an item because it must wait for the sequence to allow late randomization on the item. The sequencer can't use the item until the sequence calls `send_request` at which point the item is ready to be consumed. Even if the item is ready within the sequencer, the driver may not be ready. And when the driver is ready, the sequence may want to deliver an updated item, and as the item is passed via reference, it allows for updating the sequence item right before it gets driven on the interface of the driver. After send_request, which is a function, returns, the sequence is not allowed to update the request. Late randomization is happens in the time between wait_for_grant and send_request.

iii. Response item

As of today, a response item can be sent to the sequence via the sequencer using one of the four methods:

```
item_done(rsp)
seq_item_port.put(rsp)
seq_item_port.put_response(rsp)
rsp_port.write(rsp)
```

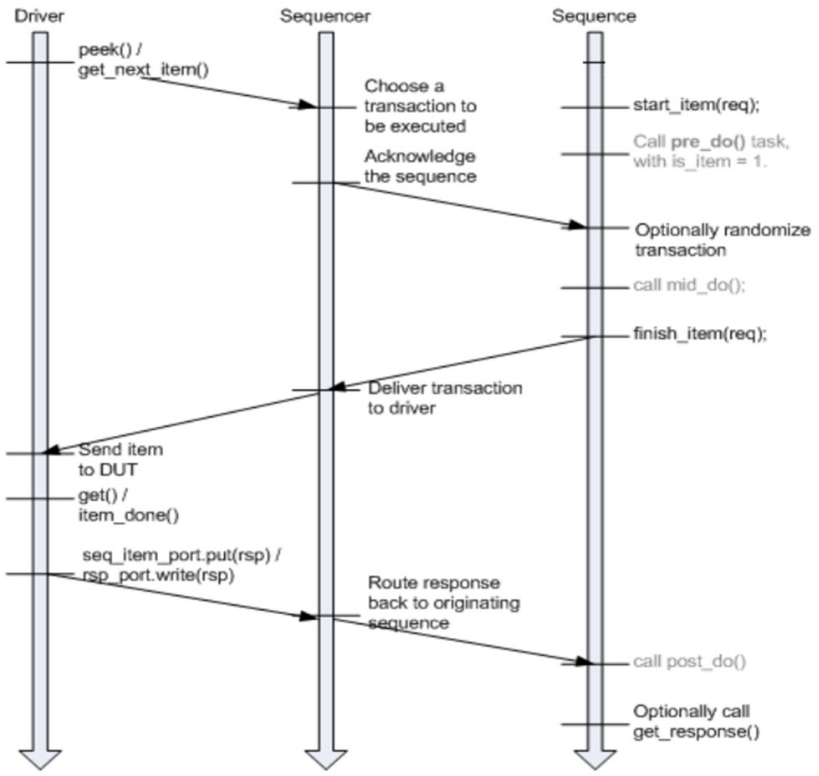

Figure 2: Flow of sequence item in pull mode [1]

III. PROPOSED TLM2 PROTOCOL

In this section, we describe phases between an initiator and a target for the proposed TLM2 protocol. Here is the state diagram for our proposed TLM2 protocol:

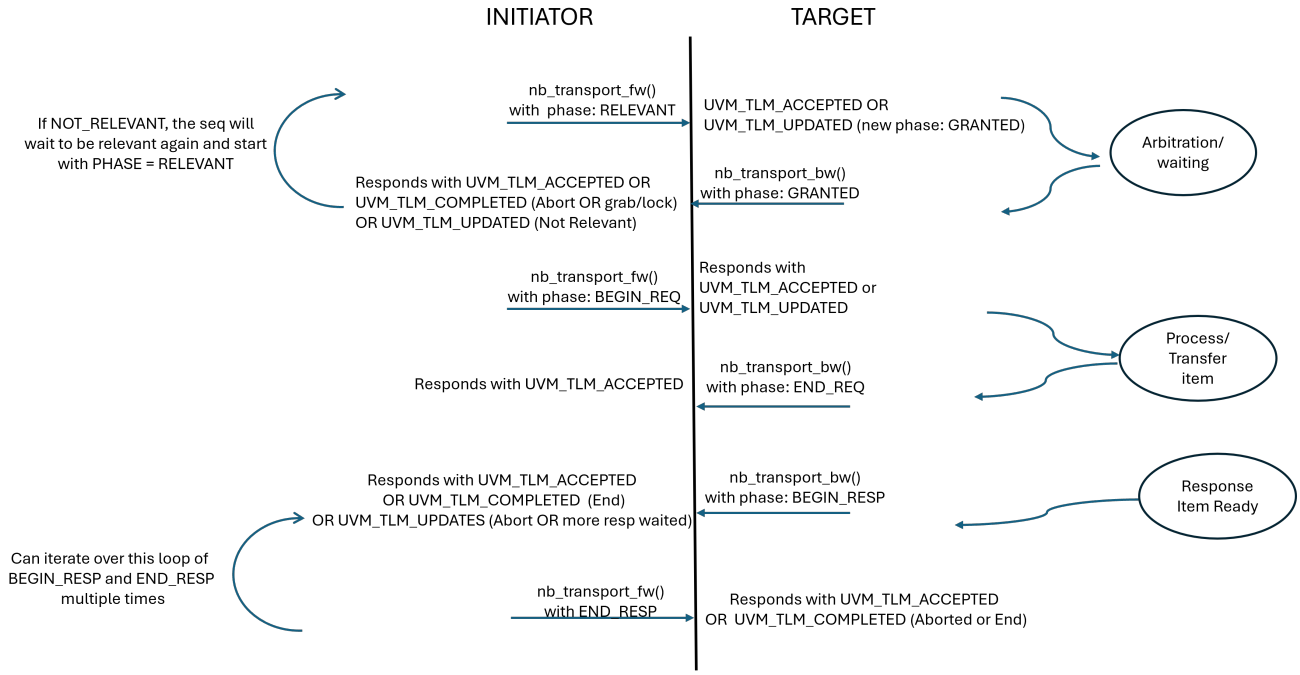

Figure 3: Proposed phases between an Initiator and a Target

As per Figure 3, this protocol has three stages:

i. *Arbitration Stage* – This is the stage where Initiator lets the Target know that it is ready to create/send a transaction using the phase RELEVANT. If the Target is immediately ready to process, it returns with UVM_TLM_UPDATED and updates the phase to GRANTED. However, the Target may take its own time to arbitrate, wait for the reset, or do other time-consuming activities that may delay its readiness. In that case it responds with UVM_TLM_ACCEPTED. Once the Target is ready, it updates the Initiator with the phase GRANTED. The Initiator can respond with UVM_TLM_ACCEPTED, UVM_TLM_COMPLETED or UVM_TLM_UPDATED (in case the Initiator is no longer relevant). If the Initiator is no longer relevant, it goes back to the waiting stage for becoming relevant again and initiating the handshake from the top.

ii. *Request Transfer Stage* – In this stage the Initiator is ready to create and/or transfer the item to be processed to the Target. It sends out the item using the BEGIN_REQ phase. The Target will respond with UVM_TLM_ACCEPTED or UVM_TLM_UPDATED. Once the Target accepts the item, it can process it, which could be sending it out on the bus, or to the next Target, or any other time-consuming activity and then once done, let the Initiator know using the END_REQ phase. The Initiator will respond with UVM_TLM_ACCEPTED. The Initiator may not consume time between the Arbitration Stage and the BEGIN_REQ phase.

iii. *Response Transfer Stage* – If there are responses involved, then the Target lets the Initiator know of a response item that is ready with the BEGIN_RESP phase. Initiator will respond with UVM_TLM_ACCEPTED or UVM_TLM_UPDATED (in case it wants to Abort or End) or UVM_TLM_COMPLETED (if it is end of transaction). The Initiator will then process the response item and will let the Target know via END_RESP phase. The Target will then respond with UVM_TLM_ACCEPTED or UVM_TLM_COMPLETED (In case of Abort or End). Depending on the protocol, you could iterate over this stage multiple times until all the responses are all received.

Other things to note:

i. *More Phases* – There are two other phases that are required to map the communication completely. ABORTED phase is required to let the Initiator or the Target know that the handshake has been Aborted. The Initiator could also let the Target know that it is in NOT_RELEVANT phase, which means the Target needs to hold the spot till the Initiator becomes relevant.

ii. *Using non-blocking transport calls* - As we see above, communication between the sequence-sequencer-driver requires a handshake, especially during the arbitration stage. A blocking transport call will not be able to accommodate such a bi-directional handshake. Hence, we use non-blocking transport calls for implementation of TLM2 protocol.

For mapping the communication between sequence-sequencer-driver, in one case, the Initiator is the sequence, and the Target is the sequencer. And in the other case, the Initiator is the sequencer, and the Target is the driver.

IV. TLM2 FOR SEQUENCER AND DRIVER

The new protocol in Section III can express all the behavior of communication between the sequencer and the driver. Hence, simple translation gaskets can be created between the sequencer and the driver - The Driver Gasket and the Sequencer Gasket. This implementation does not interfere with the existing implementation of the UVM driver or the UVM sequencer. It serves two purposes:

1. By placing the gaskets back-to-back between uvm_sequencer and uvm_driver, we can prove that the TLM 2.0 APIs are capable of handling all current sequencer/driver communication.
2. If a TLM2 based sequencer/driver pair is supported in the future (see section V), then these same gaskets provide a means of maintaining backward compatibility with older implementations.

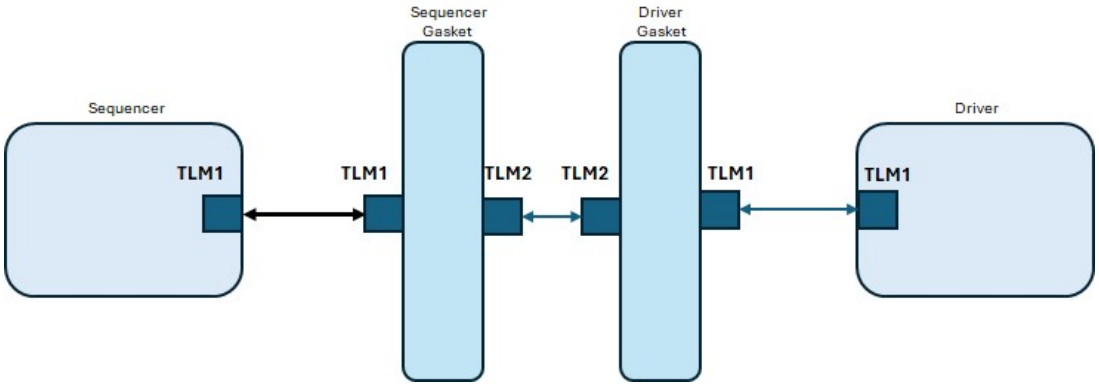

Figure 4 : Breaking down the communication between uvm_sequencer and uvm_driver into TLM2 protocol using gaskets.

Mapping the protocol in Section III to the TLM2 interaction between the Sequencer Gasket and the Driver Gasket shown in Figure 4 above, this is how the interaction looks like:

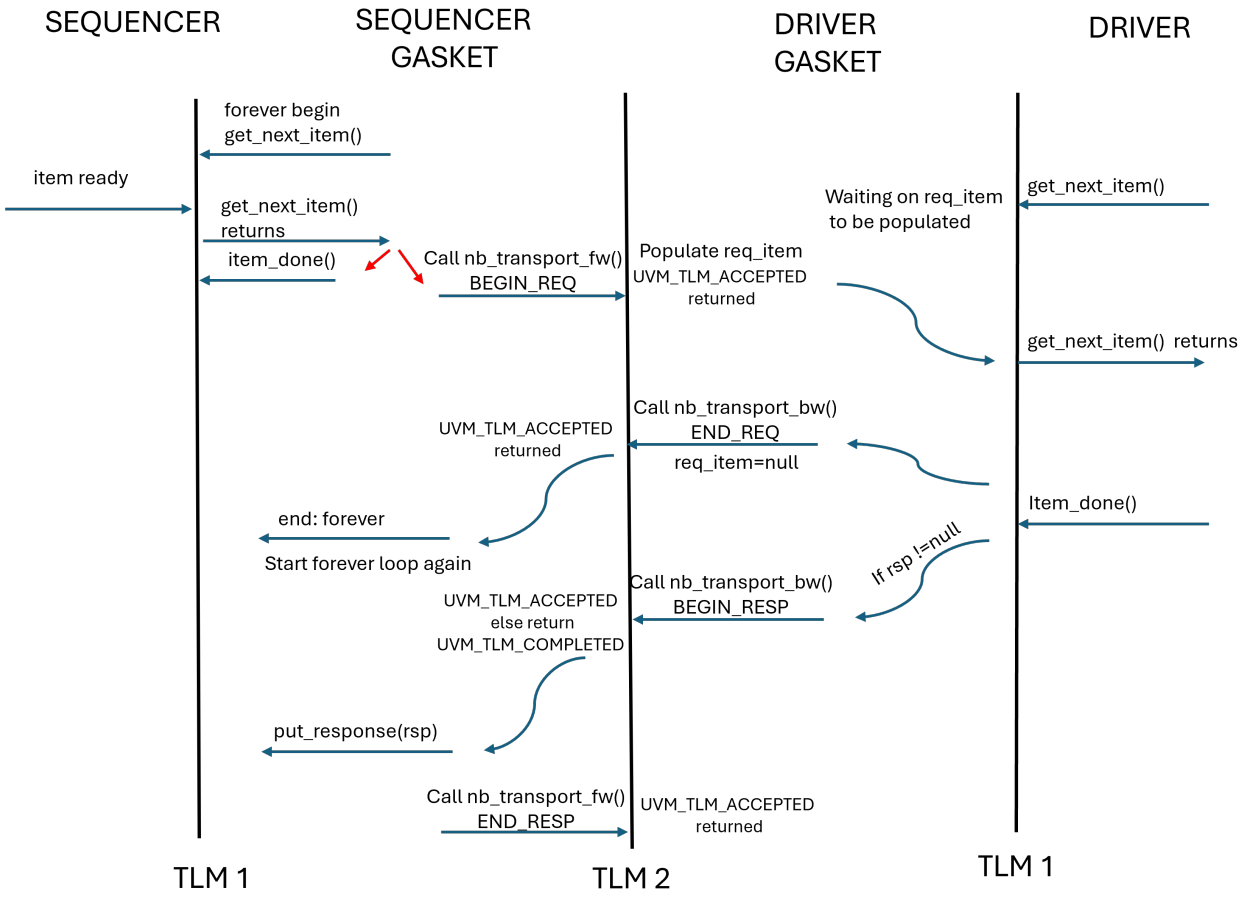

Figure 5: TLM2 based protocol between a Sequencer and a Driver using gaskets

A. Driver Gasket

The driver gasket connects to the driver using the TLM1 protocol and connects to the sequencer gasket using the TLM2 protocol. The goal of this gasket is to behave like a TLM1 sequencer for the TLM1 driver and to behave as a TLM2 driver for the TLM2 interface of the sequencer gasket. To behave as a TLM1 sequencer, it consists of

seq_item_pull_imp and the implementations for sqr_if functions. To behave as a TLM2 driver for the TLM2 sequencer, it has a target socket and an implementation of nb_transport_fw.

All the functions of the seq_item_pull_imp convert the communication to TLM2 protocol. For instance, when get_next_item is called, it does not return until the req_item is populated through a nb_transport_fw call on the target socket initiated by the sequencer gasket. Similarly, when item_done or put_response is called with a response, it triggers a nb_transport_bw call on the sequencer gasket to send the rsp to the sequencer.

B. Sequencer Gasket

The sequencer gasket connects to the sequencer using the TLM1 protocol and connects to the driver gasket using the TLM2 protocol. It consists of a seq_item_pull_port to act as TLM1 driver for the sequencer. And it has an initiator _socket and an implementation for nb_transport_bw to implement the TLM2 protocol.

In the run_phase of the sequencer gasket, there is a forever loop, which calls get on the sequencer. Once that request is populated, it triggers a nb_transport_fw call on the driver gasket to populate its req_item and unblock the driver's get_next_item call. Once the driver calls item_done, it initiates nb_transport_bw with an END_REQ. That completes the forever loop for the sequencer gasket indicating that this request is served and it can now begin requesting for the next item. When a response is received from the driver either via put_response or via item_done, it initiates a nb_transport_bw call in the driver gasket, triggering a put_response in the sequencer via the seq_item_port, sending the response to the sequencer.

As shown in Figure 5, the sequencer-driver handshake for get_next_item and item_done can be completely handled by TLM2 without the need of any additional APIs or tweaks.

Similarly, try_next_item can be handled by returning the value of req_item, instead of waiting for it to be populated. The return value could be null, or the actual sequence item.
Looking at other pull methods in the driver:

```
peek()   = get_next_item();
get()    = get_next_item();
           item_done();
```

To summarize, we are able to map the entire communication between the sequencer and the driver completely to a TLM2 based protocol.

V. SEQUENCE AND SEQUENCER

To map the communication between a sequence and a sequencer on a TLM2 protocol, we must accommodate bi-directional handshake, late randomization and mapping to TLM2 phases. The sequence-sequencer handshake introduces additional complexity, because of the presence of lock requests and relevancy checks other than the standard sequence item exchange.

A. Using TLM2 for the protocol to exchange a standard sequence_item

We first address the basic protocol between sequence and the sequencer with wait_for_grant, send_request and wait_for_item_done. To address the payload-less handshake between the sequence and the sequencer, we create a common container class. The object of this class is used as the payload across all handshakes. This container class contains the sequence item when generated, other argument variables required for wait_for_grant, send_request and wait_for_item_done. This resolves our concern about handshakes without a payload. This container class extends from the uvm_sequence_item and contains all the argument variables required for this multi-step communication.
However, given there are multiple back and forth between the sequence and the sequencer before we expect to see any response, it doesn't make sense to use existing TLM2 phases, which allow for multiple back and forth using BEGIN_RESP and END_RESP, but do not allow for multiple back and forth for BEGIN_REQ and END_REQ. Hence, we create more TLM2 phases to accommodate for this back and forth before we start seeing responses.

The user will send item of type uvm_tlm2_sequence_payload #(REQ,RSP) across the TLM2 sockets. User's sequence item will be a property of this class that will be sent across.

```
class uvm_tlm2_sequence_payload #(type REQ=uvm_sequence_item, type
RSP=REQ) extends uvm_sequence_item;
    REQ    req_item;
    RSP    rsp_item;

    //Add variable for the three calls
    int pri_req = -1; // priority for wait_for_grant
    bit lock_request = 0; // lock request for wait_for_grant
    bit rerandomize = 0; // rerandomize request for send_request
    int transaction_id = -1; // transaction id for wait_for_item_done

    function new(string name = "uvm_tlm2_seq_item");
        super.new(name);
    endfunction
endclass
```

New TLM2 phases enable us to convert the communication between the sequence and the sequencer to a complete TLM2 based protocol.
Proposed TLM2 phases:

```
typedef enum {
    UVM_TLM2_SEQ_RELEVANT,
    UVM_TLM2_SEQ_NOT_RELEVANT,
    UVM_TLM2_SEQ_GRANTED,
    UVM_TLM2_SEQ_BEGIN_REQ,
    UVM_TLM2_SEQ_END_REQ,
    UVM_TLM2_SEQ_BEGIN_RESP,
    UVM_TLM2_SEQ_END_RESP,
    UVM_TLM2_SEQ_ABORTED
} uvm_tlm2_seq_phase_e;
```

The protocol between the sequence and the sequencer can now be completed via a TLM2 based protocol. The existing functions and tasks within the sequence now call `nb_transport_fw` methods and rely on the phase value from `nb_transport_bw` methods to complete their tasks.

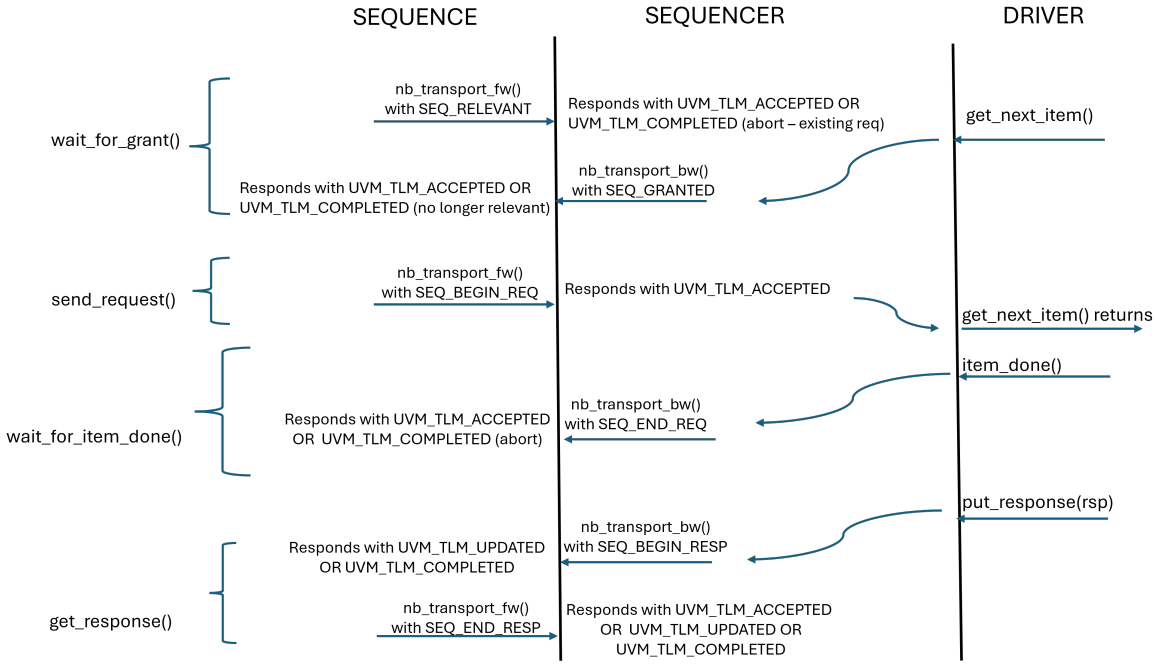

Figure 6: TLM2 protocol between sequence and sequencer using common container class and new phases

B. Using TLM2 for grab/lock interactions between the sequence and the sequencer

The sequencer's lock and grab mechanisms, which allow a sequence to gain exclusive access to the driver, can also be mapped to a TLM 2 compliant communication protocol. This provides a more robust and transparent method for handling these control-flow interactions, similar to how standard item transport is handled.

Like the previous section A on standard item, the lock/grab interaction could be modeled using custom phases within the generic payload we created earlier, uvm_tlm2_sequence_payload, as suggested by the protocol in Figure 7 (or any similar, externally defined specification). The sequencer receives the payload with the specific phase UVM_TLM2_SEQ_LOCK_GRAB via the `nb_transport_fw` path, process it, and sends a response back to the sequence once finished via `nb_transport_bw`, confirming that it has been granted exclusive control.

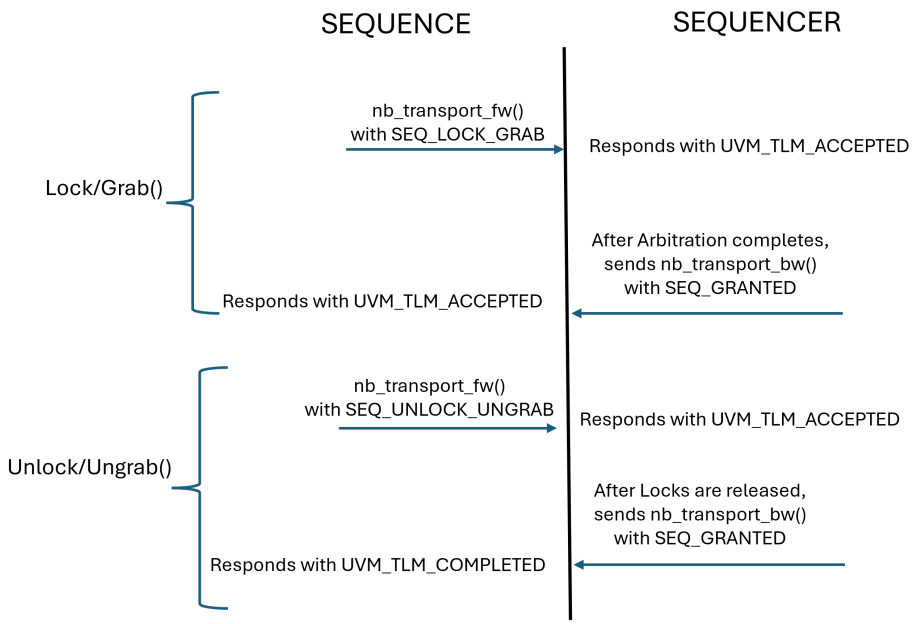

Figure 7: TLM2 protocol between sequence and sequencer for grab/ungrab and lock/unlock

C. Alternative TLM2 implementation

As an alternative to the container class solution suggested above in A and B, we can choose to stay within the predefined TLM 2.0 phases by using additional payloads to indicate the various stages of the handshake. We create a special payload class for the handshake before request and use another payload class for after the request is generated. This enables us to use the existing TLM2 phases and communication now requires two separate payload classes.

In this protocol, we use two different items to complete our sequence and sequencer handshake. The `wait_for_grant` uses the uvm_tlm2_grant_item to communicate with the sequencer. The `send_request` and `wait_for_item_done` use the uvm_tlm2_sequence_payload to communicate. Both the handshakes can be mapped to the existing TLM2 phases.

Both the approaches stated above are functional and correct, it's a subjective decision on what is more palatable: Sticking to the TLM standard phases, but requiring higher level knowledge of REQ meanings, or extending TLM standard phases, but having all knowledge of the handshake contained within a single data object.

Our objective here was to establish that the communication between the sequence and the sequencer can be mapped to TLM2 based protocol.

VI. CONCLUSION

The current communication based on TLM1-like protocol is complicated and raises concerns due to its quasi-non-blocking behavior, and its lack of ability to abort or otherwise indicate the end of handshakes [2],[3],[4].

To map the sequence-sequencer interaction that exists today, you cannot use TLM2 as-is, you must either modify the payload or modify the TLM2 phases. Today's existing generic versions do not fit the complex needs of the interaction.

Any blocking API between the sequence-sequencer and the sequencer-driver that exist today, can be mapped to TLM2 non-blocking APIs. Therefore, there is nothing about the sequence-sequencer-driver interactions that mandates a blocking API.

A non-blocking, TLM2-expressable API has been presented in this paper that's capable of representing everything that the current solution can do, plus it resolves all of the problems from the current TLM1-like implementation.

TLM2 will bring in more standardized communication protocols, interoperability and the ability to assign responses to requests automatically. This can be further leveraged to communicate multiple responses per request. TLM2 drivers can also be connected to TLM2 initiators. Thus, bringing in more interoperability.

Implementation of the gaskets described in section IV and tests are available in GitHub [6],[7].

REFERENCES

[1] IEEE Std 1800.2TM, IEEE Standard for Universal Verification Methodology Language Reference Manual, 2020.
[2] IEEE Std 1800TM, IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language, 2012.
[3] Lies My Teacher Told Me About The UVM – Basic Stimulus, SNUG Boston, 2015
[4] Universal Verification methodology (UVM) 1.2 User's Guide, Accellera
[5] OSCI TLM 2.0 Language Reference manual, Accellera
[6] https://github.com/nv-negoyal/uvm-core/tree/tlm2
[7] https://github.com/nv-negoyal/uvm-tests/tree/tlm2