UVM Transaction Recording Enhancements

Rex Chen (rex_chen@springsoft.com), Bindesh Patel (bindesh_patel@springsoft.com), Jun Zhao (jun_zhao@springsoft.com)

Research & Development, SpringSoft, Inc., Hsinchu, Taiwan

Introduction

SystemVerilog provides a compelling advantage in addressing the verification complexity challenge — not simply as a new language for describing complex structures, but as a platform for driving a more efficient, realistic test of the design. It is no surprise then that the adoption of the language for verification purposes has been rapid. Along with this growth, methodologies like VMM/OVM have been developed to help verification engineers to create testbenches efficiently and with maximum reuse and a consistent well-documented methodology. However, the need for interoperability among verification libraries has become a critical issue for users to integrate testbench code (usually in the form of Verification IP) developed using different methodologies. This has motivated a new methodology, Universal Verification Methodology (UVM), to be developed under the auspices of Accellera for the expressed purpose of fostering universal verification IP interoperability. Led by electronics companies and supported by a suite of companies representing the breadth of the verification ecosystem, the UVM will increase productivity by eliminating expensive interfacing that slows verification IP reuse.

At the outset, UVM provides the infrastructure for a transaction recording scheme using so-called "hooks" functions that can be implemented by the user or a third-party to record the transactions occurring in the testbench into some database. This could be as sophisticated as a specialized debug database or as simple as a text file. While this provides utility for debug, there are some limitations in the current transaction recording scheme provided in UVM. Firstly, the current recorded data is not enough for a more complete and efficient debugging view. Secondly, the automatic recording for non-sequencers is clearly lacking when compared to what is possible for sequencers. Users or third-parties who want to overcome these limitations have to modify the overall UVM library which causes unnecessary intrusion on the user's flow. So we propose a convenient non-intrusive enhancement to UVM, so that users or third-parties can utilize the enhanced recording without having to touch the UVM library. For better visibility and efficiency, we propose more debugging information to be recordable for each transaction. We also propose a better scheme to record transactions for non-sequencers.

This paper proposes these enhancements for UVM transaction recording. It further explores potential visualization and debug front-ends to leverage a more complete captured trace.

Keywords

SystemVerilog, UVM, OVM, Transactions, UML, Sequence Diagram

THE CURRENT UVM TRANSACTION RECORDING SCHEME

A verification environment built on top of UVM does not provide any capability to record transactions in any format by default. However, it does provide a “hooks functions” capability for third parties or users to record the data to some specific format. These related hooks functions are listed below:

- uvm_create_fiber
  Given a name, create a stream with transactions.
- uvm_set_index_attribute_by_name
  Not used.
- uvm_set_attribute_by_name
  Add a named attribute to this transaction. This is invoked by uvm_component::end_tr(). For monitor and driver, user has to call this API manually.
- uvm_check_handle_kind
  Return the type of handle. Legal types are 'Stream', `Fiber` and 'Transaction'.
- uvm_begin_transaction
  Return a handle to a new transaction. The new transaction has a variety of properties:
  - It belongs to a 'stream‘
  - It has a name
  - It starts either now or at begin_time, if begin_time is non-zero
- uvm_end_transaction
  Given an open transaction handle, end it. If end_time is non-zero, then end the transaction at end_time.
- uvm_link_transaction
  Given two transaction handles, create a "relationship" between them
- uvm_free_transaction_handle
  Given a transaction handle, release storage for it. Calling free_transaction_handle() means that the handle is no longer to be used anywhere.

These functions are empty in the original uvm_misc.sv file. That is, there is no implementation and are meant for users or third parties to implement as per their requirements. Vendors or users can determine how to store the recorded transactions. For example, if a user wants to record the transactions into a text file, he can implement the following functions as listed:

- uvm_create_fiber:
  Generate and return the stream handle. Log the stream name into the text file.
- uvm_set_index_attribute_by_name:
  Call uvm_set_attribute_by_name().
- uvm_set_attribute_by_name
  Log the attribute name, value, radix, and number of bits into the text file.
- uvm_check_handle_kind:
  Return the type of handle. The type of handle could be stream or transaction.
- uvm_begin_transaction
  Log the transaction name and begin time of transaction into the text file.
- uvm _end_transction:
  Log the transaction name and end time of transaction into the text file
- uvm_link_transaction:

Log the names of two transactions and their relationship into the text file.

- uvm_free_transaction_handle

  Log the freed transaction handle.

Usually these function implementations are written into a SystemVerilog file. Let's say the file name is "hooks.sv".

Now, to enable automatic recording of transactions, the user would have to:

1. Include the transaction hook file "hook.sv" in uvm_misc.sv.
2. Enable transaction recording in the "build" function of test by adding:

   set_config_int("*", "recording_detail", UVM_FULL);

3. Compile the design and run simulation

This so-called "hooks" mechanism only records transactions from sequencers automatically when users turn on the recording scheme. To record the transactions from other components like drivers or monitors, users have to add uvm_component::begin_tr() and uvm_component::end_tr() in their testbench code.

Figure 2 shows the waveform-view realization of recording UVM transactions into a debug database using the current recording scheme provided.

Disadvantages of the Current UVM transaction recording Capabilities

Based on the current recording scheme provided in UVM, the following information is recorded:

- Stream (the collection of related transactions) name, scope
- Transaction name, begin and end time
- Attribute name and value
- Relation between sub-sequence and sequence

While this information is useful, the current recorded information is not sufficient for efficient debugging. Let's illustrate this using the recording result of the XBus example in the UVM library. Figure 1 shows the architecture of the Xbus example.

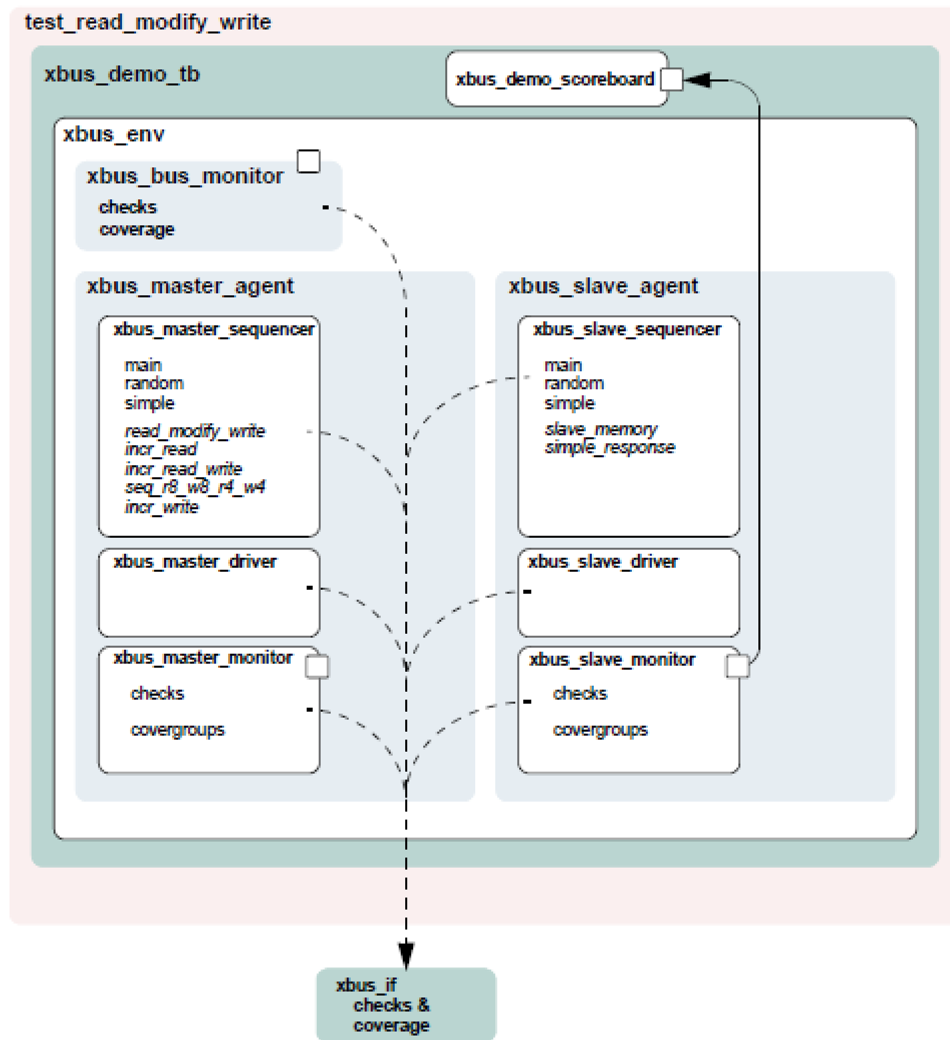

Figure 1: XBus Demo Architecture

The xbus_env component contains any number of XBus master and slave agents. In the example shown in Figure 1, the xbus_env is configured to contain just one master and one slave agent. The xbus_master_ and xbus_slave_agent are structured identically with the only difference being the protocol-specific function of its subcomponents.The XBus master agent contains up to three subcomponents: the sequencer, driver, and monitor. By default, all three are created.

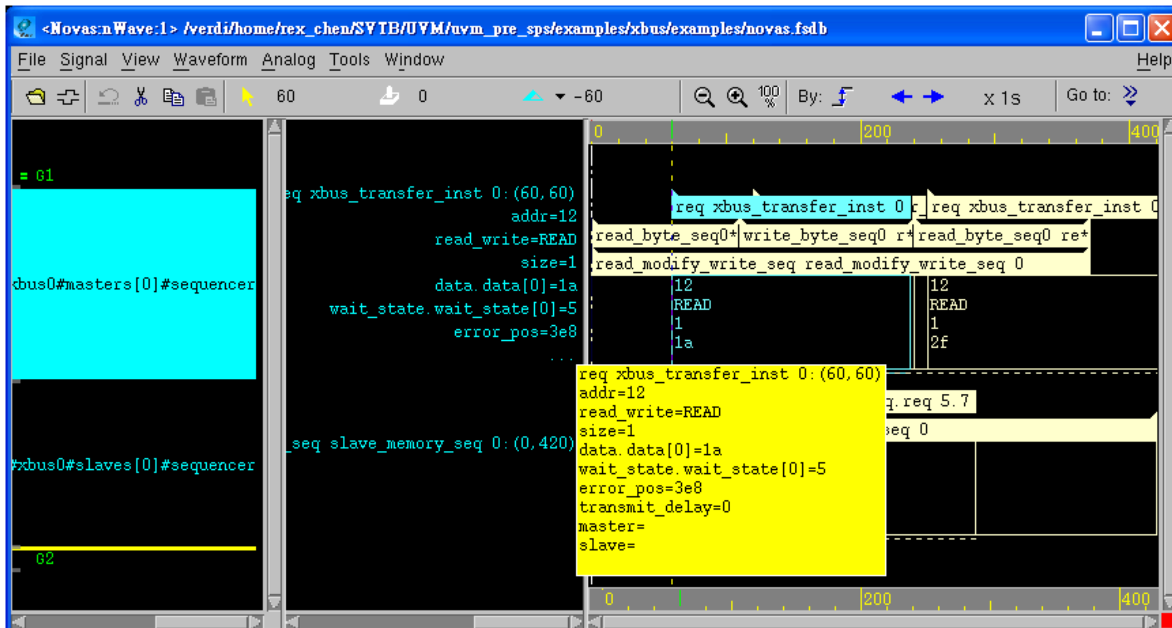

Figure 2: The recording result of XBus example before enhancements

Figure 2 shows the transaction recording result of the XBus example before the enhancements that will be proposed later in this document. The "xbus[0]#master[0]#sequencer" stream contains the sequence "read_modify_write_seq" because the component "xbus[0]#master[0]#sequencer" sends the sequence "read_modify_write_seq". The sequence "read_modify_write_seq" consists of three sub-sequences: "read_byte_seq0", "write_byte_seq0", and "read_byte_seq0". As such, they have a parent-child relationship. Each sub-sequence contains one sequence item "req". The attributes and values of sequence item "req" are all recorded and displayed as atrributes. For the "req (60,60)" sequence item, it's begin and end time is 60 and 60 respectively. The attributes of "req(60,60)" have the following values:

- addr = 12
- read_write = READ
- size = 1
- data.data[0] = 1a
- state.wait_state[0] = 5
- error_pos = 3e8
- transmit_delay = 0
- master =
- slave =

The stream "xbus[0]#slave[0]#sequencer" contains sequence "slave_memory_seq" because the component "xbus[0]#slave[0]#sequencer" sends the sequence "slave_memory_seq". There is a sequence item "req" in it.

While this recorded data is useful, it is not complete. Referring to Figure 2:

1. We do not know the destination component of the sequence item or sequence. To make it complete, we propose to record the name and type of initiator/target component.

2. We do not know which sub-sequence the sequence item "req" belongs to. Therefore, we suggest binding the full sequence path name to it. For example, "read_modify_write_seq.read_byte_seq0.req" is the full sequence path name of "req (60,60)".
3. There is no relation information recorded between sequence items and sub-sequences. For example, we cannot tell whether the sub-sequence "read_byte_seq0" consists of one or more sequence items. Therefore, we propose that each sequence item or sequence should have a unique id. Then sequence item or sub-sequence can use an id array to describe the hierarchy relation. For example, we can see seven sequence items and sequences on stream "xbus[0]#master[0]#sequencer". Let's say the unique ids of "read_modify_write_seq", "read_byte_seq0", "write_byte_seq0","read_byte_seq0", "req", "req" and "req" are "1","2","3","4","5","6", and "7" respectively. The id array of the three "req" sequence items are "1.2.5","1.3.6" and "1.4.7". So we know that the three sequence items "req" belong to different sub-sequences, but they belong to the same sequence "read_modify_write_seq".
4. An attribute is needed to indicate if the sequence item is a response item from driver.

Besides the completeness of the data discussed above, another consideration is that in the current scheme, only the sequence items or sequences from sequencers are recorded automatically. Therefore, we cannot see the sequences from non-sequencer components. In this example, we cannot see sequences from the monitor unless we add begin_tr() and end_tr() in the monitor class manually. Figure 3 shows the recording result of the sequencer and monitor after we insert begin_tr() and end_tr() manually in the monitor class.

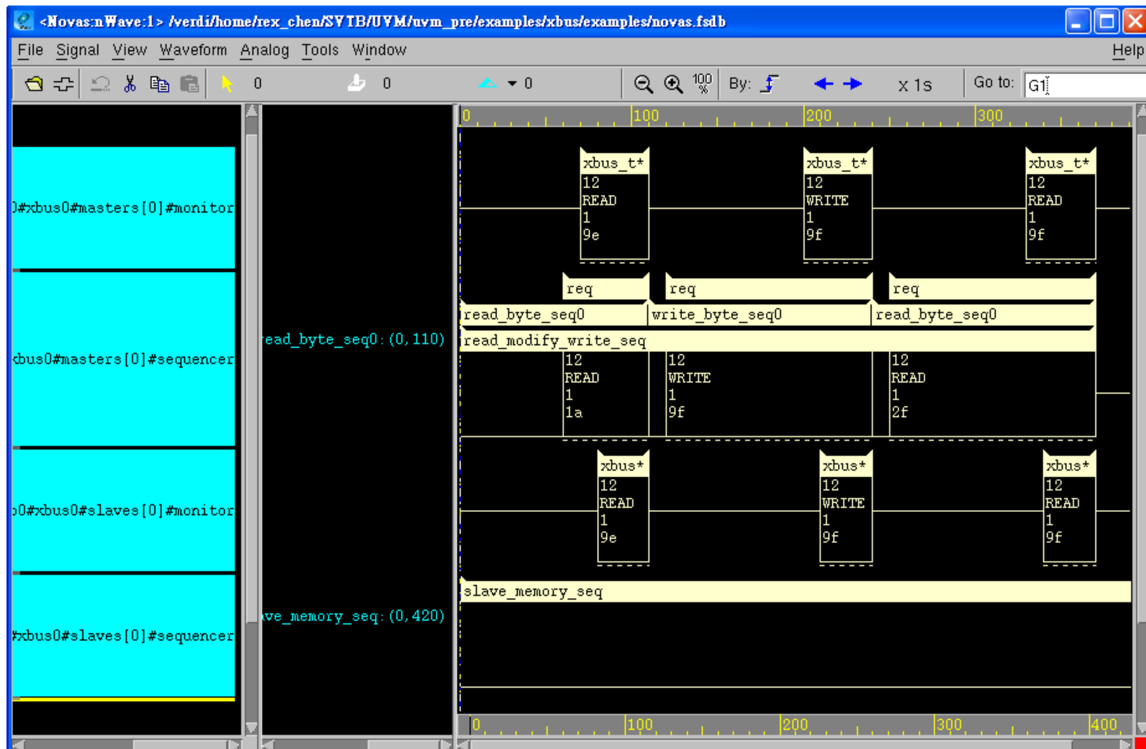

Figure 3: the recording result of sequencer and monitor

ENHANCEMENTS FOR UVM TRANSACTION RECORDING

As discussed, the current recording scheme does not provide sufficient information for efficient debugging. Additionally, to record non-sequencer components, users have to call uvm_component::begin_tr() and uvm_component::end_tr() manually. If users forget to call one of them, the recorded transactions will be incomplete. Therefore, we need a better scheme to enforce the pairing of begin_tr()/end_tr(). We will next propose enhancements to address these two drawbacks with the current scheme.

To enhance the UVM transaction recording, we propose the following additional information to be recorded for each transaction:

- The component name that the transaction comes from.
- The component type that the transaction comes from.
- The component name that the transaction goes to. There could be multiple destinations.
- The component type that the transaction goes to.
- The full sequence path name of the transaction.
- The id array that corresponds to the full sequence path name.
- Whether the transaction is response item from driver.
- The unique id of transaction.

With this additional information, users will have a more complete database of the traffic between the testbench components.

For non-sequencer components, we propose that users only be required to add begin_tr() in their testbench code. The end_tr() will be called automatically in the port function that sends the transaction. For example, the user usually calls uvm_analysis_port::write() to send transaction in monitor. The end_tr() should be called in this function and it would check if the begin_tr() is called or not. If users forgets to call begin_tr(), the port function will call it automatically. In this case, the time could be incorrect. Even with this drawback, this mechanism is still better than the original one in that at least the data will be recorded (rather than nothing being recorded). Users can always add begin_tr() in their testbench codes to make it right.

Figure 4 shows the waveform realization of the recording result after the proposed enhancements are put into effect.

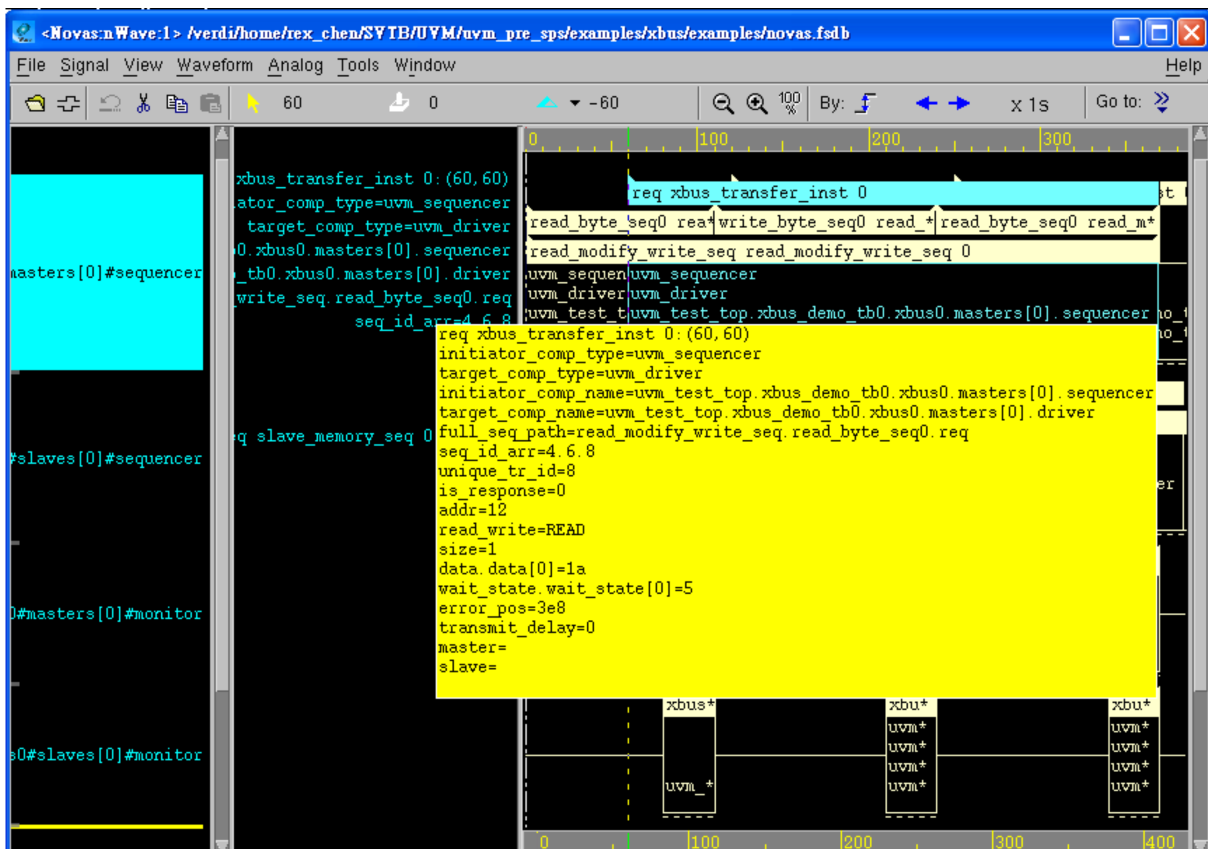

Figure 4: The recording result for XBus example after enhancements

As can be seen, the following additional attributes are recorded after the enhancements:

1. initiator_comp_type = uvm_sequencer (The initiator component type)
2. target_comp_type = uvm_driver (The target component type)
3. initiator_comp_name = uvm_test_top.xbus_demo_tb0.xbus0.masters[0].sequencer (The initiator component name)
4. target_comp_name = uvm_test_top.xbus_demo_tb0.xbus0.masters[0].driver (The target component name)
5. full_seq_path = read_modify_write_seq.read_byte_seq0.req (The full sequence path name)
6. seq_id_arr = 4.6.8 (The id array of the corresponding full sequence path name)
7. unique_id = 8 (The unique id of this sequence item)
8. is_response = 0 (This is not a response item)

From the enhanced recorded data, we can tell that the sequence item "req (60,60)" is sent from the sequencer "uvm_test_top.xbus_demo_tb0.xbus0.masters[0].sequencer" to driver "uvm_test_top.xbus_demo_tb0.xbus0.masters[0].driver". Its full sequence name is "read_modify_write_seq.read_byte_seq0.req". Therefore, this sequence item belongs to sub-sequence "read_byte_seq0". The sub-sequence "read_byte_seq0" belongs to sequence "read_modify_write_seq". The id array "4.6.8" indicates that the ids of "ead_modify_write_seq","read_byte_seq0" and "req" are "4", "6" and "8" respectively. This sequence item is not a response item from the driver. Clearly this is a more complete record of the activity and provides the user better visibility into the testbench.

ENHANCED VISUALIZATION

Comparing the waveform views before and after the proposed enhancements (Figure 2 and Figure 4), we can see that the additional attributes are indeed recorded after the proposed enhancements are implemented. However, the waveform view of the transactions does not provide a clear picture of the component traffic. The additional attributes can only be viewed in a text format.

For a better more natural realization of the recorded data, we can derive inspiration from the Unified Modeling Language (UML) and specifically its sequence diagram specification. While the UML sequence diagram is primarily used for documentation for software systems, there has been research in using it to visualize the execution of programs for which a trace has been recorded [2]. With the additional recorded attributes, it becomes possible to realize a sequence diagram type of view for better visualization of the traffic in a SystemVerilog testbench.

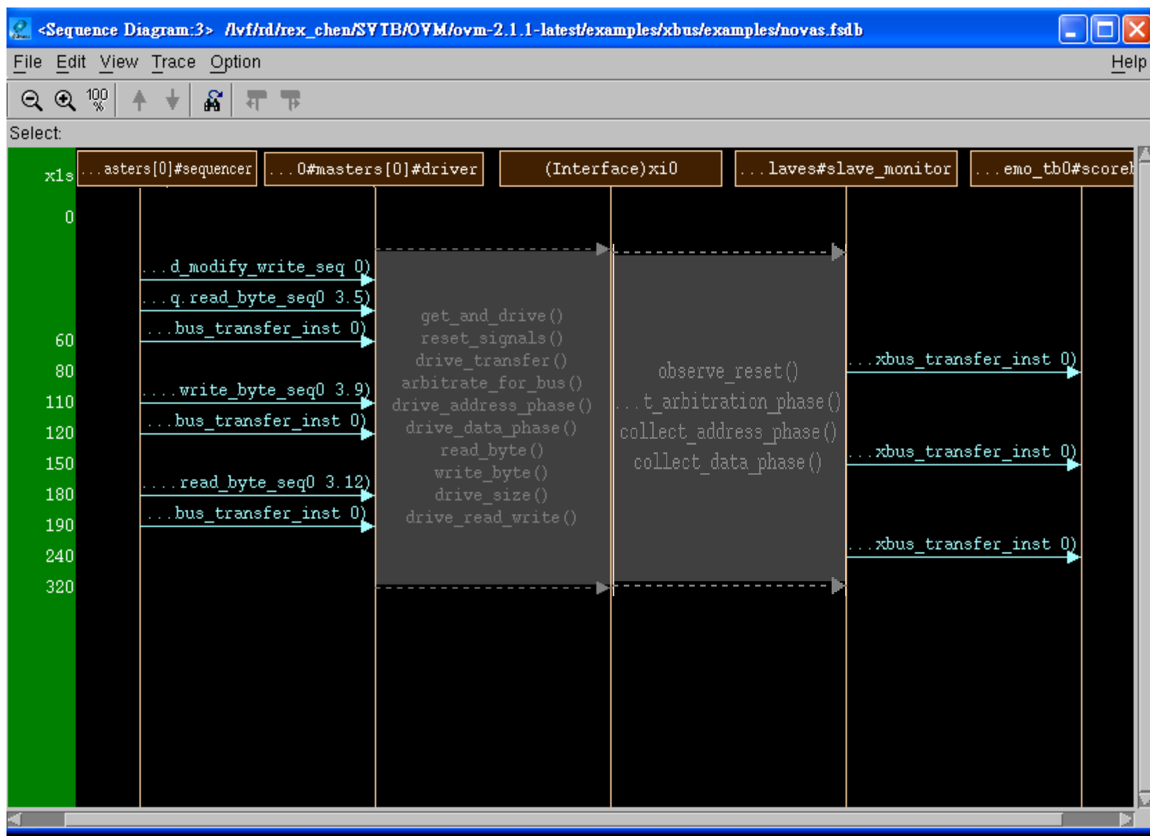

Figure 5: The sequence diagram for XBus example

Figure 5 shows the sequence diagram for the XBus example. The vertical line represents simulation time. The brown rectangles on the top of the sequence diagram are the components in the testbench. In this example, we can see the "sequencer", "driver", "interface", "monitor" and "scoreboard" components.. The blue arrows are the sequence items or sequences between components. Therefore, we can see that the sequencer sends "read_modify_write_seq", "read_byte_seq0", "bus_transfer_inst(req)", "write_byte_seq0", "bus_transfer_inst(req)", "read_byte_seq0" and "bus_transfer_inst(req)" to the driver. Likewise, the monitor

sends three sequence items “xbus_transfer_ints” to the scoreboard. Mapping the blue arrows to the time line, we can figure out the begin time of each sequence item or sequence.

Since the sequence items and sequences from sequencers have a hierarchical relationship, we can collapse the sequence items into sub-sequences. Figure 6 shows the collapsed result.

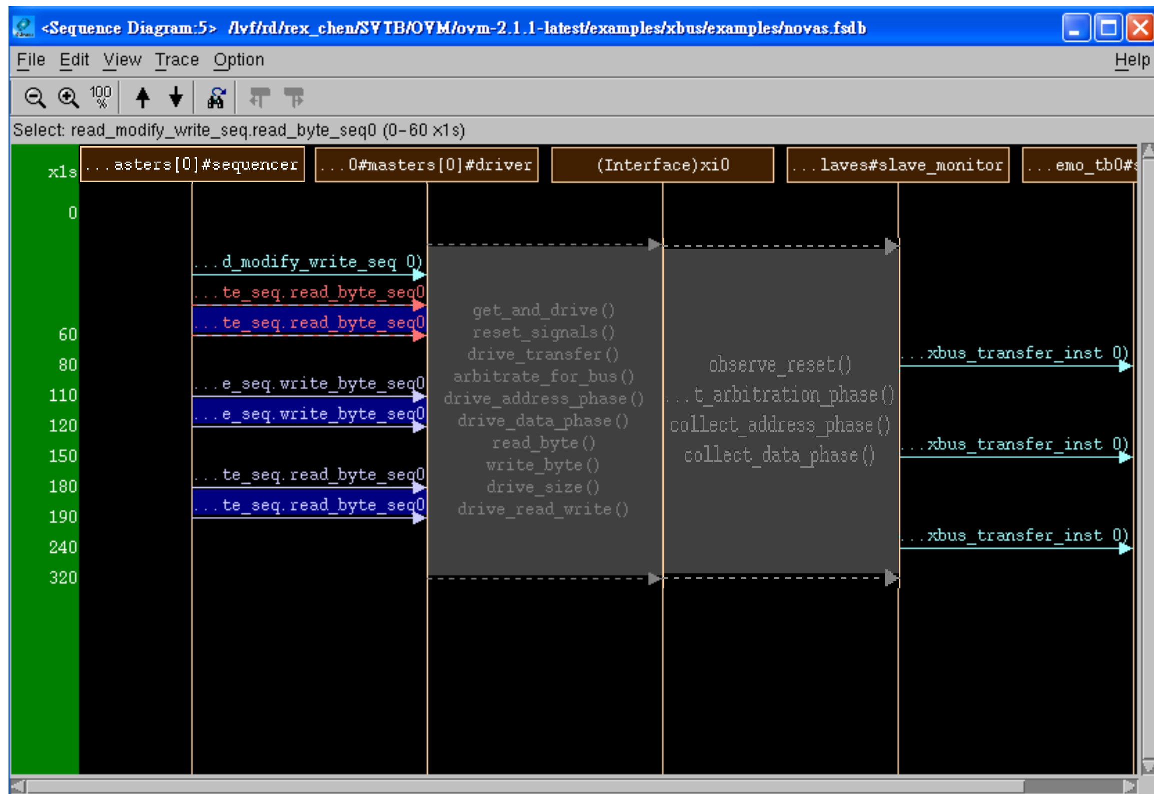

Figure 6: Collapse the sequence items into sub-sequence

The grey areas between the driver/interface and interface/monitor are possible functions to transform sequence items (transactions) to signals or vice versa. Since we have no dynamic data for the transformation functions, the only way to derive this information is to analyze the source code to find the possible functions. In this example, the possible transformation functions in the driver class are get_and_drive(), reset_signals() ....drive_read_write(). The sequence items are transformed to signals and sent to the interface using these functions. On the other side, the functions obseve_reset(), collect_arbitration_phase(), collect_address_phase() and collect_data_phase() in the monitor class collect the signals from the DUT and transform them into sequence items (transactions).

Conclusion

UVM provides a transaction recording scheme which is important for visualization and debugging of transaction-level traffic in the testbench, which in turn helps with the debug of the testbench and DUT. However, the recorded information is not sufficient and it is not convenient to record data for non-sequencer components. We propose an enhanced scheme which involves recording additional

attributes and providing a better mechanism for transaction recording of non-sequencers. Moreover, with the additional recorded attributes, we introduce a sequence diagram view to show the testbench traffic more clearly and naturally.

REFERENCES

1. UVM User Guide and Reference Manual, *http://www.accellera.org/activities/vip*
2. OVM User Guide and Reference Manual, *http://www.ovmworld.org/resources.php*
3. Katharina Mehner and Bernd Weymann. Visualization and Debugging of Concurrent Java Programs with UML. *Dissertation, University of Paderborn, February 2005*
4. Katharina Mehner. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. *Revised Lectures on Software Visualization International Seminar, May 2001.*
5. Object Management Group. Unified Modeling Language. *http://www.uml.org/*