Extendable Messaging Techniques for Debugging and Analyzing UVM Testbench Structure and Transaction Flow

Jun Zhao, Bindesh Patel, Rex Chen
Research & Development
Synopsys, Inc.
Mountain View, California and Hsinchu, Taiwan

***Abstract*— This paper first outlines current debug capability embedded within the UVM library and then proposes additions to this capability that can significantly add visibility and debugability into the execution of the testbench as an integral part of the entire environment. Ideas for presenting the additional acquired debug information to the user are also introduced.**

Keywords— UVM; SystemVerilog; testbench; debug; debugging; tracing message; transaction; recording

I. INTRODUCTION

The Universal Verification Methodology (UVM) has now established itself as the standard methodology of choice for improving verification efficiency and data portability, including reuse and interoperability. The methodology includes a SystemVerilog class library that allows users to efficiently build realistic transaction-based testbenches.

The raw SystemVerilog language does not provide any standard mechanisms for recording simulation activity for the class-based structures used in UVM as it does for HDL signals and nets with the callback-driven Verilog Procedural Interface (VPI) standard. It can be further argued that even if such an application programming interface (API) exists, the low-level data returned would be of limited use in today's high-level verification environments in which the transaction is the atom of data. In fact, the library itself is turning out to be the best candidate for acquiring useful debug data at the appropriate abstraction level.

Ideally, a debugging system should be able to be attached to the UVM Class Library as a library extension, so that it can dynamically access and process the internal testbench data during simulation. Unfortunately, the UVM Class Library does not provide an open mechanism to allow the development of reusable extensions. It does provide so-called callback functions, but these are actually virtual functions of a base class. The virtual function-based callbacks are restricted and only the extended class can make use of these callbacks. In other words, the implementations of the callbacks can be only developed inside a specific testbench. They are not library extensions and, therefore, cannot be reused in other testbenches.

In this situation, the system messages inside the UVM Class Library become very important for users in order to understand or debug the testbench execution. Currently, there are some system messages in the UVM library, for example, tracing messages for phasing/objection, and auditing messages for config_db/resource_db. However, these are not enough to cover all the key UVM functions and important stages of the testbench execution. Without system messages as tracing points, it is virtually impossible to collect any useful dynamic testbench data.

In this paper, we are going to discuss the benefits that can be derived from inserting some additional system messages inside the UVM Class library. We have experimental data from adding system messages on two major aspects of the UVM: the testbench structure and transaction flow. The paper will further illustrate how to capture the data from UVM system messages directly into a debug database and how to make use of the database to improve the efficiency of post-simulation analysis and debug. Enhanced visualization applications can be built on top of the collected data to better understand and debug the UVM testbench simulation.

II. CURRENT UVM DEBUGGING CAPABILITIES AND LIMITATIONS

The UVM Class Library [1] provides the building blocks needed to efficiently develop well-constructed, reusable verification components and test environments using the SystemVerilog language and especially relies on its object-oriented syntax and semantics. While the UVM is trying to strike a balance on reuse, portability, and encapsulation of the verification components for testbenches, the issues of extensibility and debug have yet to be fully addressed.

Extensibility, in this case, means the UVM should provide open interfaces so that library extensions can be developed by vendors or CAD groups and attached to the UVM library. During simulation, all the library extensions could be closely combined with the UVM library and passively join the UVM execution. These extensions could transparently collect and process the dynamic data and monitor the flow in UVM-based testbenches, without modifying the testbench behavior. The library extension should be separate from the original UVM library, and there should be no need to modify any original

code inside the UVM library. The library extension should also be separate from the user's UVM-based testbench, such that it is attached to the UVM library and can be reused by any UVM-based testbenches. The testbench developers do not need to have any knowledge about the library extensions. Note that fundamentally, extensibility is one aspect of reusability. Here we are focusing on library extensible and reusable capabilities.

One analogy of such extensibility is the Verilog PLI (Programming Language Interface) as an open simulation interface. The PLI allows users to extend Verilog by creating user-defined system tasks and registering user-defined hook functions, such that the hook functions are called when a signal value or any other simulation state changes. The hook functions can collect many current-state values from the simulation and can return values back to simulation. The Verilog PLI has been the key technology to facilitate the post-simulation debugging for HDL designs, without which we would be stuck in the stone age of interactive simulation debugging. The waveform of value changes for each signal in the HDL design can be recorded into a database for further processing and visualization using the Verilog PLI technology. Unfortunately, this technology does not work well for the object-oriented part of the SystemVerilog language, which is more like a software language. The SystemVerilog Testbench (SVTB) class variables are dynamic data and therefore cannot be dumped.

However, library extensions as previously discussed would greatly facilitate a proper debugging system for the UVM library and UVM-based testbenches. For example, testbench developers or users may want to log or record the important details of UVM execution to help with testbench comprehension or post-simulation debugging. Currently, they have to manually instrument debugging code within the testbench code or inside the library which is generic to all UVM-based testbenches. They are forced to duplicate these debug instrumentation in all the testbenches, or alternatively, modify the UVM library and put the code inside the library. The modified UVM library will, however, have the portability issue.

In fact, the current UVM library does include a minimal set of useful features for debugging purpose. For example, a transaction recording scheme [2] is provided to record UVM sequence behavior and contents into preferred database via uvm_recorder. The uvm_recorder maintains a set of virtual functions that are originally empty but can be re-implemented by extending the uvm_recorder and replacing the uvm_default_recorder. These so-called "hook" functions in uvm_default_recorder are automatically called at the key stages of sequence generation, and the transaction timing, payload and layering information can be recorded through the "hook" functions. It is up to the user's implementation of "hook" functions how, where and what information is recorded, which leaves space for extensibility. While it is a good utility for debug, the transaction recording only covers one part (sequence generation) of the UVM at a fairly high level. It is of no help when the user wants to look at the transaction transitions across the UVM component hierarchy or between two verification components.

Tracing messages are another mechanism provided by the UVM library to dump debug information, similar to debugging messages that can be output from any software system. These messages are embedded inside the UVM library at the major points of the execution, enabling important runtime data to be observed, collected and printed to the screen or a log file. Users can turn on each portion of the tracing messages by activating them from the command line options. The following lists the command line arguments provided by UVM to turn on these tracing messages:

+UVM_PHASE_TRACE	turns on tracing of phase executions
+UVM_OBJECTION_TRACE	turns on tracing of objection activities
+UVM_RESOURCE_DB_TRACE	turns on tracing of resource DB access (read & write)
+UVM_CONFIG_DB_TRACE	turns on tracing of configuration DB access

These tracing messages turn out to be a very convenient and efficient mechanism for debugging. Nevertheless, there are two major drawbacks: 1) it does not cover all the functionalities of the UVM; 2) it can only be output to a text format log file, which is difficult for post processing. We will address these two issues in the following two sections.

III. Adding New Tracing Messages into the UVM Class Library

For the first issue of limited tracing messages being recorded, the resolution is quite simple: add new tracing messages into the UVM library. We propose that tracing messages be added at least in the following categories of UVM functionalities:

- Trace how the component hierarchy is built and how the ports/sockets are connected;
- Trace the UVM factory registration and override configuration;
- Trace the traffic at the TLM1 port interface and capture the pass-through transactions, requests and responses, etc.;
- Trace the TLM2 socket interface and capture the pass-through transaction (the generic payload), sync, phase, and basic protocol, etc.; and
- Trace the register access (read and write, mirror, etc) and how the register hierarchy has been built.

In this paper, we will discuss how to add new tracing messages in order to observe testbench structure generation (including component creation and port connection) and monitor the transaction flow at the port level.

A. Tracing Component Creation and Port Connection

The testbench structure includes the component hierarchy and the TLM (transaction layered modeling) port connections. By adding system messages at the build phase, users are able to trace how the components (including ports, which are also components) are created and record the parent-child relationship between the components. Further, by inserting system messages at the connect phase, users can also trace how the TLM ports (including TLM2 sockets) are connected and record the producer-consumer relationship between the connected ports.

The following methods are the points at which tracing messages are added for component and port creation:

```
function uvm_component::new (string name,
                             uvm_component parent);

function uvm_port_base::new  (string name,
                             uvm_component parent,
                             uvm_port_type_e port_type,
                             int min_size=0,
                             int max_size=1);
```

The following information is collected and passed as additional fields for the message:

- The parent full name
- The component/port name
- The type name, e.g., “ubus_pkg::class ubus_master_driver”
- Other component/port info (e.g., is_port, is_export, is_imp, etc.)

The following methods are the points at which tracing messages are added for port or socket connection:

```
function void uvm_port_base::connect (this_type provider);
```

The following information is collected and passed as additional fields for the message:

- The caller port full name and port type, etc.
- The provider port full name and port type, etc.

Example 1 illustrates how tracing messages are added for component/port creation and port connection:

```
function uvm_component::new (string name,
                             uvm_component parent);
  …
  // Add a message whenever a new component has been
  // created. The port component will be reported when
  // creating the uvm_port_base, so won't be reported here.
    begin
      uvm_port_component_base port_component;
      if (!$cast(port_component,this))
        `uvm_info ("COMP_TRACE",
                   {"Creating  component ",
                    (parent==top?"":
                   {parent.get_full_name(),"."}),name,
                    "(type=",get_type_name(),")"},
                    UVM_LOW)
    end
endfunction

function uvm_port_base::new (string name, …);
  …
  // Add a message whenever a new base port component
  // has been created.
  `uvm_info ("PORT_TRACE", {"Creating port ",
             m_comp.get_full_name(),
             " (type=",get_type_name(),")"}, UVM_LOW)
endfunction

function void uvm_port_base::connect (this_type provider);
  …
  // Add a message whenever two ports are connected.
  `uvm_info ("PORT_CONN_TRACE", {"Connecting ports ",
             this.get_full_name(),
             " with ",provider.get_full_name()}, UVM_LOW)
endfunction
```

Example 1

B. Tracing Transaction Flow at the Port Level

The transaction (or UVM sequence) flows are from component to component through TLM ports. Adding system messages at the TLM port-level enables users to observe how transaction data is transferred from one port to another and what type of methods are used to transfer the data. Here the data is not from the transaction point of view, but from the perspective of port-to-port interface protocol by way of port method calls. The system messages record what method of a

certain port is called at a certain time, when it finishes, what is the content of the data passing to the method, what is the return value, etc.

The following methods add the tracing messages for TLM1 port and TLM2 socket interface:

TLM1 Ports:

```
task put (TYPE arg);
function bit try_put (TYPE arg);
function bit can_put();
task get (output TYPE arg);
function bit try_get (output TYPE arg);
function bit can_get();
task peek (output TYPE arg);
function bit try_peek (output TYPE arg);
function bit can_peek();
task transport (REQ req_arg, output RSP rsp_arg);
function bit nb_transport (REQ req_arg,
                           output RSP rsp_arg);
```

Sequence Item Pull Ports:

```
task get_next_item(output REQ req_arg);
task try_next_item(output REQ req_arg);
function void item_done(input RSP rsp_arg = null);
function void put_response(input RSP rsp_arg);
task get(output REQ req_arg);
task peek(output REQ req_arg);
task put(input RSP rsp_arg);
```

Analysis Ports:

```
function void write (input T t);
```

TLM2 Sockets:

```
function uvm_tlm_sync_e nb_transport_fw (T t, ref P p, input uvm_tlm_time delay);
function uvm_tlm_sync_e nb_transport_bw(T t, ref P p, input uvm_tlm_time delay);
function task b_transport (T t, uvm_tlm_time delay);
```

The following information is collected and passed as additional fields for the message:

- The request and/or response transactions
- The return value if any
- The method name, e.g., “put”, “get”, etc.
- The times entering and leaving the method
- The port info (full name, type, configurations, etc.)
- The generic payload, phase/sync, and delay for TLM2

Example 2 shows how tracing messages for transaction flow are added at the sequence item pull port level. The methods used in this example are get_next_item() and item_done(). The uvm_report_record() is a function created to print out the messages. Later in this paper we will demonstrate how this function can also be used to dump data into a database.

```
// A container class that wraps the data to be recorded.
class uvm_port_recording_object extend uvm_object;
    uvm_port_component_base port_comp;
                              // The base port component handle
    string func_name;         // The port interface method name
    uvm_object req;           // The transaction payload
    time begin_time;          // The begin time of the method call
    time end_time;            // The end time of the method call
endclass

// The macro to be added at the beginning of each port
// interface method. It initiates the container object and records
// the beginning time.
`define UVM_IF_METHOD_BEGIN \
    uvm_port_recording_object port_value = new; \
    port_value.begin_time = $time;

// The macro to be added at the ending of each port interface
// method. It records the method name, the base port
// component,  the ending time, and the transaction payload.
// The uvm_report_record() method will call the UVM
// transaction recording hook functions and record the data
// into database.
`define UVM_IF_METHOD_END(req_arg,method_name) \
    port_value.func_name = method_name; \
    port_value.port_comp = m_comp; \
    port_value.end_time = $time; \
```

```
    if ($cast(port_value.req,req_arg)) \
      uvm_report_record ("PortIF",
        "Port level recording …",    port_value);


// Add the macros to each TLM or sequence port method
`define UVM_SEQ_ITEM_PULL_IMP(imp, REQ, RSP, req_arg, rsp_arg) \
    task get_next_item(output REQ req_arg); \
      `UVM_IF_METHOD_BEGIN \
      imp.get_next_item(req_arg); \
      `UVM_IF_METHOD_END(req_arg,"get_next_item") \
    endtask \
    function void item_done(input RSP rsp_arg = null); \
      `UVM_IF_METHOD_BEGIN \
      imp.item_done(rsp_arg); \
      `UVM_IF_METHOD_END(rsp_arg,"item_done") \
    endfunction \
```

Example 2

IV. Saving UVM Message Data Into a Database

The second issue with tracing messages as mentioned earlier is that they are only output to a text format log file. Since there can be a huge number of messages, the log file can be extraordinarily large and very unwieldy in terms of organizing and processing the data, or even locating useful data. For debug and analysis purposes, users typically want to record as much data as possible to narrow down problems. The solution is to save the messages into a well-organized database and build a good user interface on top of the database to retrieve and visualize the data. This requires a database format that can easily save message data with predefined properties (e.g., verbosity, severity, etc.). The message data should also be associated with any user-defined properties and their values in different data types, which will help record transactions and their payloads. A set of PLI tasks that help record the data during simulation and UVM execution can be embedded into UVM testbenches or the UVM library.

PLI tasks to record the information into a database can be placed wherever the previously-discussed tracing messages have been inserted in the UVM library. However, this is not scalable and extendable, nor is it suitable for reuse. We found that the UVM Report Catcher utility can be used to hook the PLI tasks with UVM messaging without even modifying the UVM library. The uvm_report_catcher is a callback mechanism and can be used to catch messages issued by the UVM Report Server. User extensions of uvm_report_catcher (in which the action to be taken on catching the report is specified) can be registered as callbacks to capture the messages as long as the messages are issued using the UVM recommended report API or macros (e.g., `uvm_info(), etc.). Using this uvm_report_catcher facility, we implemented an extension to intercept the UVM tracing messages, and then redirect the messages into the database by using the corresponding PLI tasks. This extension does not need any user involvement except for the initialization, nor does it require any modification to the UVM library.

Saving the UVM message data into a database enables post-processing of the recorded message data and helps users to analyze and comprehend the data. Post-processing procedures include visualization, filtering, searching, ordering, and merging, etc. For example, by turning on UVM_PHASE_TRACE and UVM_OBJECTION_TRACE, the following tracing messages about phase executions and objection activities are printed out to the output screen or log file:

```
…
UVM_INFO ../../../../src/base/uvm_phase.svh(1410) @ 0: reporter [PH/TRC/SCHEDULED] Phase 'uvm.uvm_sched.pre_main' (id=378) Scheduled from phase uvm.uvm_sched.post_configure

UVM_INFO ../../../../src/base/uvm_phase.svh(1158) @ 0: reporter [PH/TRC/STRT] Phase 'uvm.uvm_sched.pre_main' (id=378) Starting phase

UVM_INFO ../../../../src/base/uvm_phase.svh(1235) @ 0: reporter [PH/TRC/SKIP] Phase 'uvm.uvm_sched.pre_main' (id=378) No objections raised, skipping phase

UVM_INFO ../../../../src/base/uvm_phase.svh(1387) @ 0: reporter [PH/TRC/DONE] Phase 'uvm.uvm_sched.pre_main' (id=378) Completed phase

UVM_INFO ../../../../src/base/uvm_phase.svh(1410) @ 0: reporter [PH/TRC/SCHEDULED] Phase 'uvm.uvm_sched.main' (id=390) Scheduled from phase uvm.uvm_sched.pre_main

UVM_INFO ../../../../src/base/uvm_phase.svh(1158) @ 0: reporter [PH/TRC/STRT] Phase 'uvm.uvm_sched.main' (id=390) Starting phase

UVM_INFO @ 0: main [OBJTN_TRC] Object uvm_test_top.ubus_example_tb0.ubus0.masters[0].sequencer.loop_read_modify_write_seq raised 1 objection(s): count=1 total=1

UVM_INFO @ 0: main [OBJTN_TRC] Object uvm_test_top.ubus_example_tb0.ubus0.masters[0].sequencer added 1 objection(s) to its total (raised from source object ): count=0  total=1

UVM_INFO @ 0: main [OBJTN_TRC] Object uvm_test_top.ubus_example_tb0.ubus0.masters[0] added 1 objection(s) to its total (raised from source object ): count=0 total=1

UVM_INFO @ 0: main [OBJTN_TRC] Object uvm_test_top.ubus_example_tb0.ubus0 added 1 objection(s) to its total (raised from source object ): count=0  total=1

…
```

For the purposes of this paper, we've displayed only a very short snippet of the messages in the log file. In contrast, if the messages are recorded into a specialized debug database, visualization applications can be used to illustrate in one single snapshot window the temporal flow or transition of UVM phasing and to display the raising/dropping/holding of UVM objections and the components that are raising/dropping/holding the objections (as shown in **Figure 1**). This goes far beyond text messages by showing the time waveform of the relevant dynamic data to further improve debug.

The UVM Report Catcher is not the only method for recording message data into a database. Modifying and replacing the report server is an alternative mechanism that can serve a similar function. However, the drawback with both these methods is that the data is transferred as a long string that must be parsed and processed before saving into the database. In **Example 2** the UVM messaging facility is not used to implement *uvm_report_record()* function. Instead, because what we want to record transaction data, it takes advantage of the UVM transaction recording mechanism and directly records the port-level transaction data through the UVM recorder. Ideally, the UVM messaging system should be enhanced to allow users to create messages that can contain additional fields and also work like transaction recording. In this way, a user-defined recorder could be hooked to the UVM library and all messages recorded into a database along with their fields.

V. POST-PROCESSING UVM MESSAGE DATA AND ENHANCED VISUALIZATIONS

As illustrated in the previous section, the visualization of phase execution and objection activities after tracing messages for UVM_PHASE_TRACE and UVM_OBJECTION_TRACE is recorded into the database. Visualization tools can be used to display the message data (currently converted from a long string into a label and list of properties or attributes) in a waveform view, which better illustrates the temporal relations between the messages. Temporal relations are more important to tracing messages, because these messages usually happen across the simulation period. Messages for different tracing purpose are put in different streams.

For example, in **Figure 1**, the tracing messages for UVM runtime phases are put in the stream of "\PH_TRC_[uvm]" and the tracing messages for UVM objections affecting the *main* phase are put in the stream of "\OBJ_TRC_[main]". Operations like search, filter and highlight are provided at the application level as these are common operations required by users. As shown in **Figure 1**, the messages related to the *main* phase are highlighted in yellow color. More advanced post process and visualization are also available. In **Figure 1**, a signal waveform (below the stream "\PH_TRC_[uvm]") is created based on the value of the *phase* attribute from all messages in the stream of "\PH_TRC_[uvm]". This phase waveform provides a clear and intuitive visual indication of the phase transitions along the simulation and the time periods of each UVM phase. Similarly,

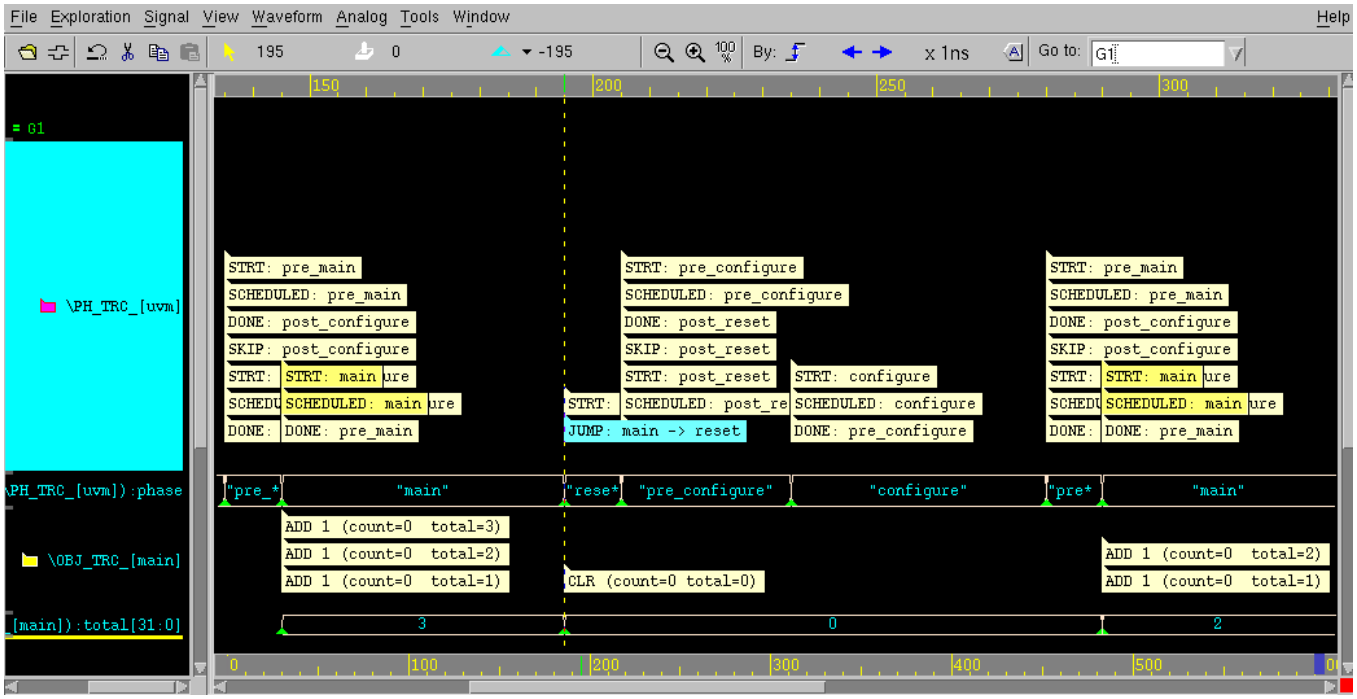

Figure 1: Waveform Illustration for Phasing Execution and Objection Activities

a signal waveform (below the stream "\OBJ_TRC_[main]") is created based on the value of the *total* attribute from all messages in the stream of "\OBJ_TRC_[main]" to visualize how the total count of objections for the *main* phase varies along the time.

Likewise, the testbench hierarchy and connection data collected from the added tracing messages can also serve to build enhanced visualization applications on top of the recorded data. Users can view the testbench hierarchy and component parent-child relationship in a tree-type illustration, as shown in the left pane of **Figure 2**. These applications

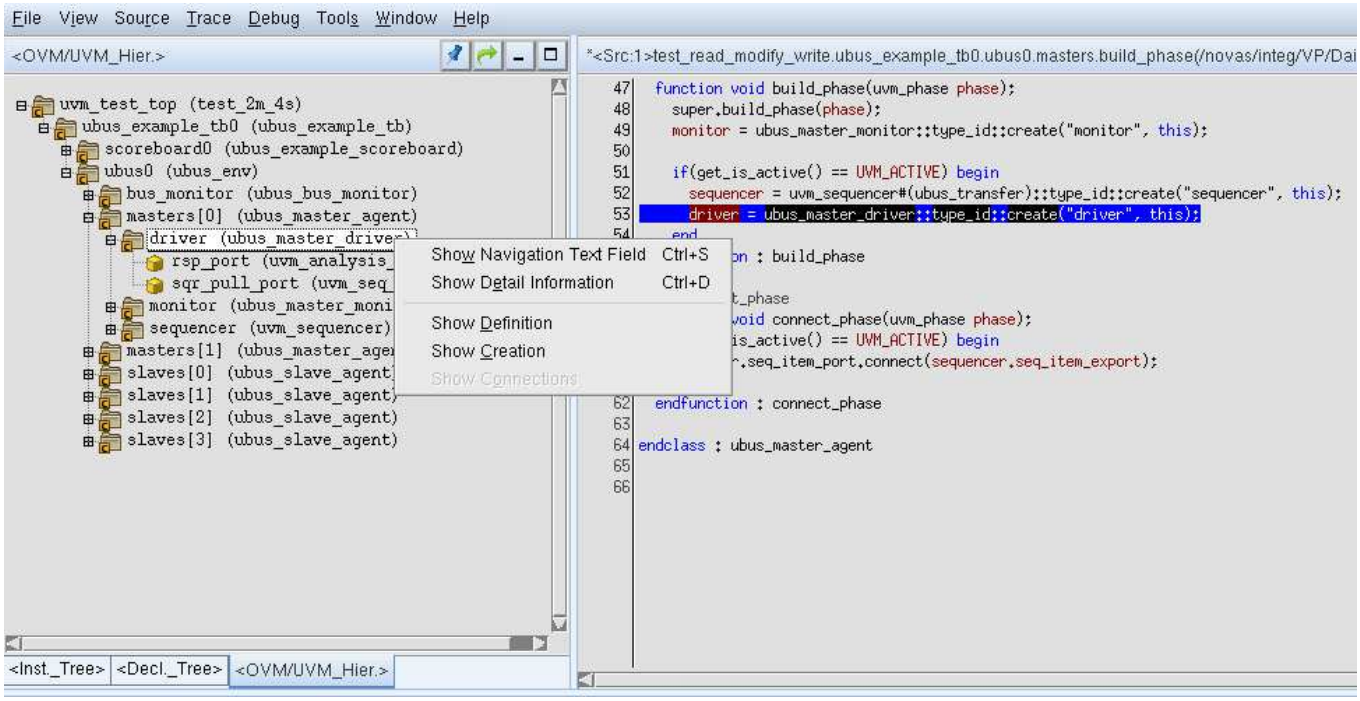

Figure 2: Illustration of UVM Component Hierarchy Tree and Source Code Synchronization

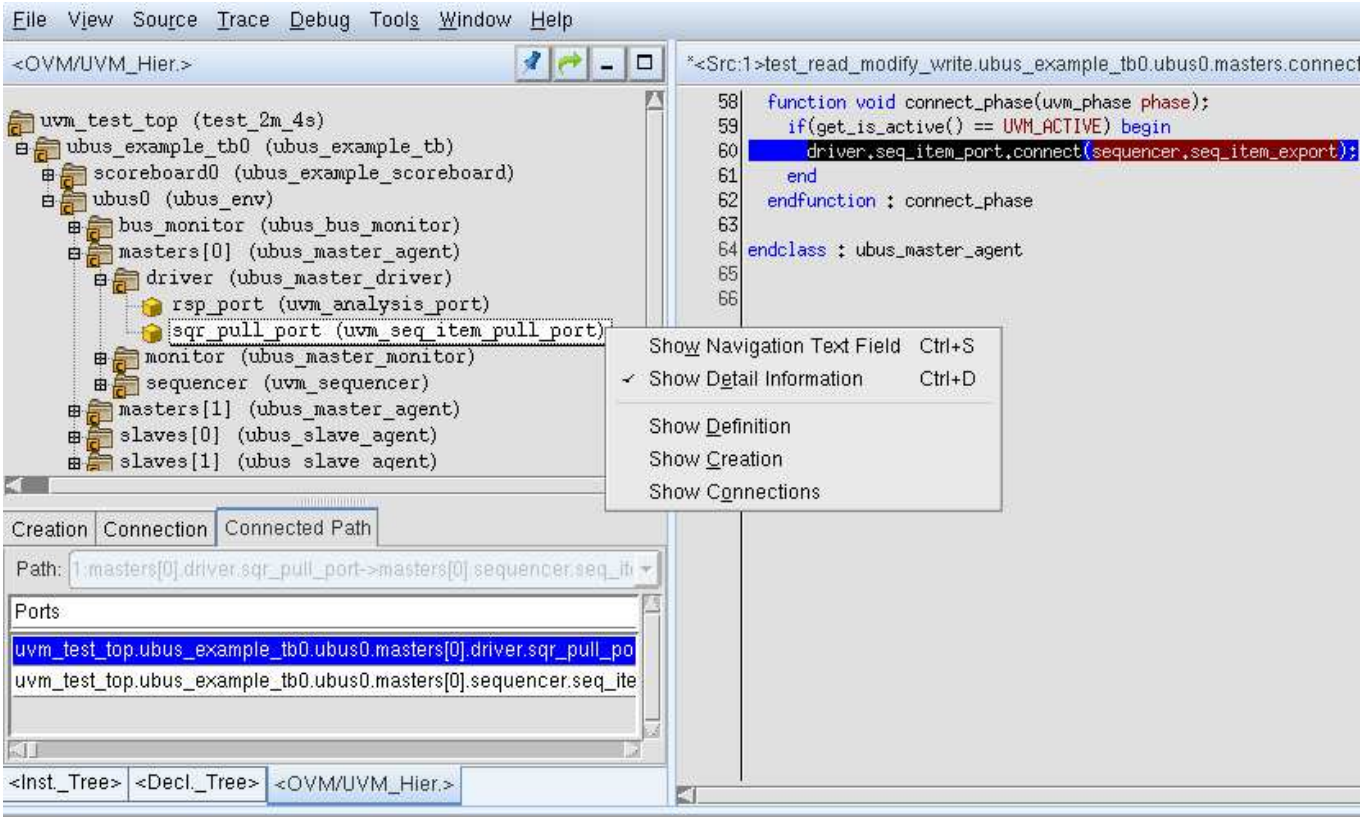

Figure 3: Displaying Ports and Port Connections in UVM Hier Tree

provide just one example for leveraging the data recorded.

The UVM Hier Tree in **Figure 2** displays the test structure of the *ubus* case with two master agents and four slave agents. The tree can be expanded or collapsed to show or hide the lower level components. The class type is also displayed beside the component name. Users can point the mouse to the component in the UVM Hier Tree and click the right mouse button (RMB) to pop up a menu, from which operations can be chosen to show the definition of the component class in the source code pane on the right (**Show Definition**), or to show where in the source code the component has been created (**Show Creation**). The right pane in **Figure 2** displays and highlights the line of the source code where the component *ubus0.master[0].driver* has been created.

Ports are also included in the UVM Hier Tree, and they are the leaves in the tree. A tab (**Connected Path**) can be opened at the bottom of the UVM Hier Tree pane to list all ports along the path that are connected to the selected port. In addition to the **Show Definition** and **Show Creation** operations, ports also have the **Show Connection** operation in the RMB menu to display where in the source code one port is connected to another port. The right pane in **Figure 3** shows and highlights the line of the source code where the port *ubus0.master[0].driver.sqr_pull_port* is connected to another port.

Transaction data is more suitable for display using a waveform viewer. Sequences and sequence items are dumped to the debug database through the UVM Transaction Recording system. In **Figure 4**, the waveform viewer displays recorded sequence data along the time meter, which illustrates the begin time and end time of the sequence, the contents of the sequence in the box, and the timing correlation information between the sequences. The parent/child relationships between the sequence and sub-sequence are displayed by appropriate highlighting, i.e., when a parent sequence is selected, the child sequences are highlighted in a bright color. Again the case is the standard UVM ubus example. Similarly, the data resulting from the tracing messages added for the transaction flow at the port level is captured and saved in the debug database. The temporal flow of transaction data through a certain port channel and the interaction between two connected ports can be illustrated by the waveform viewer, which retrieves the related data from the recorded debug database.

The top row in **Figure 4** indicates the transaction function calls happening at the sequence pull port of a UVM driver. The port interface method *get_next_item()* is called at time 60, while *item_done()* and *put_response()* are called respectively at time 110. Another *get_next_time()* call happens at time 120.

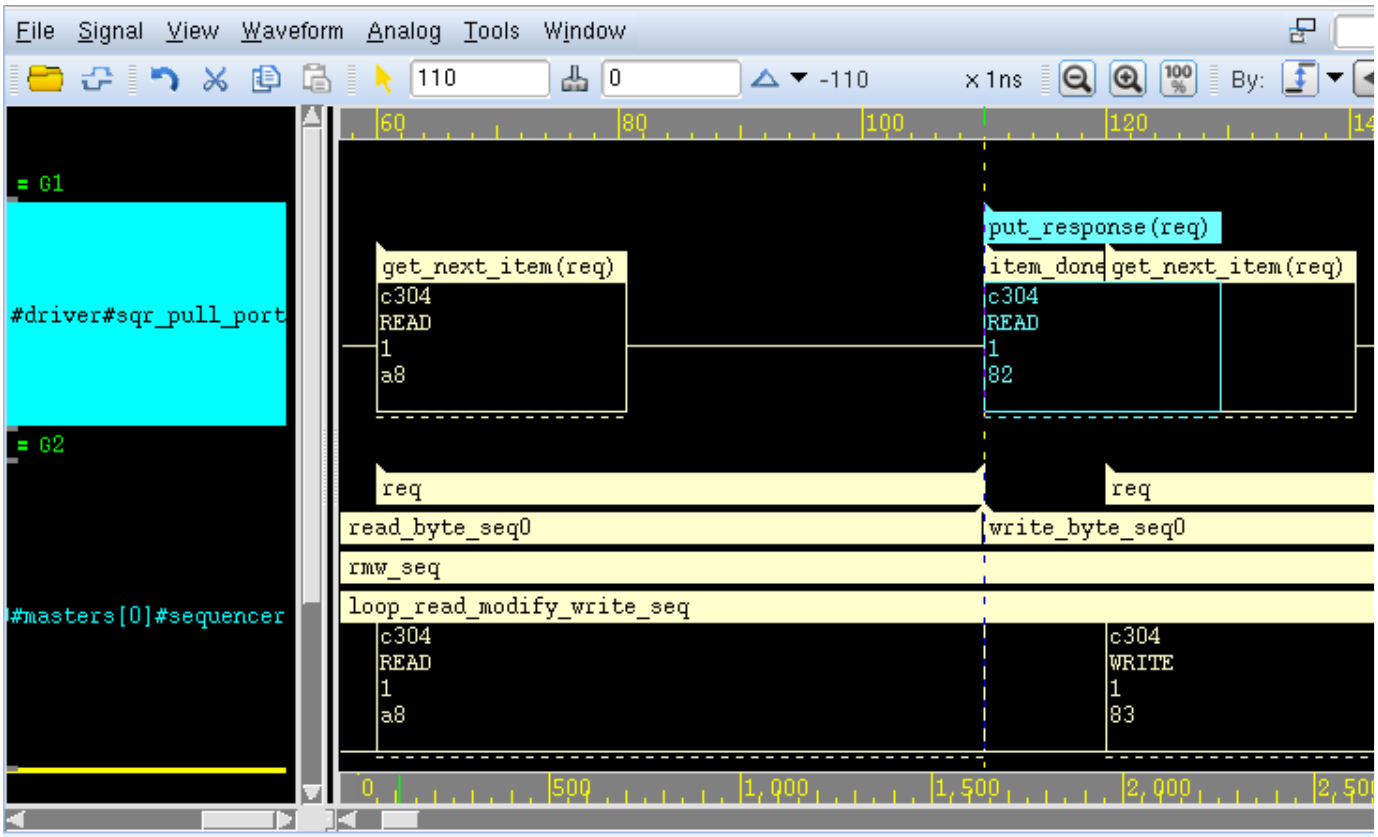

Figure 4: Illustration of Port Level Transaction Flow in Contrast with UVM Transaction Recording Waveform

This is in sync with the sequence timing shown in the bottom row and further explains the beginning and ending of the sequence. When comparing the sequence waveform at the bottom and the port level transaction flow waveform at the top, users can get a much better understanding about the transaction flow which in turn makes it easier to find and fix bugs in the design or the testbench itself. This example depicts a simple testbench. For complex testbenches, users can first analyze the higher-level sequencer transactions along with waveform techniques like zooming, filtering, highlighting, etc. to first narrow down the region of interest and then look at the detailed port-level transactions as needed.

The visualization applications for testbench structure and transaction flow can be interrelated and synchronized to further ease and streamline debug and analysis. For example, users can select a port in the hierarchy viewer and display the relevant transaction waveform. In addition, since the stack trace can be recorded through the PLI tasks as part of the message data into the debug database, the event (component creation, port connection, or transaction-level interfacing) depicted by the message can be correlated to other representations, such as source code using drag-and-drop techniques. For example, users can drag the *get_next_item()* box from the waveform viewer in **Figure 4** and drop it into the source code pane in **Figure 3** to show the source code where *get_next_time()* is called.

VI. CONCLUSION

Based on our experience, we strongly advocate that additional system messages, such as those described in the paper, should be included in the default standard UVM Class Library, so that the dynamic data at each important stage of UVM execution can be captured into a log . Furthermore, the UVM Class Library should be enhanced such that the messages can be easily captured and diverted into a debug database, which can then be used to drive a specialized UVM debug tool. Each system message would become a recording point with which internal runtime data can be actively collected and recorded into a database by reusable UVM library extensions, thereby minimizing or eliminating the burden on users to instrument directly into their testbenches or modify the UVM libraries. Further processing of the database can enable more efficient post-simulation analysis and greater understanding of UVM testbenches. It is clear that just as the Verilog PLI allowed debug to advance to a level in keeping with design complexity, so too is the critical requirement for a mechanism to record data from modern object-oriented testbenches. The mechanisms proposed in this paper make it feasible for users and vendors alike to stay in lock-step with the ever-increasing complexity that is part and parcel of any modern verification environment.

REFERENCES

[1] UVM User Guide and Reference Manual, http://www.accellera.org/activities/vip

[2] R. Chen, B. Patel, and J. Zhao, “UVM Transaction Recording Enhancements”, DVCon Proceedings, 2011