Transaction Recording Anywhere Anytime

Rich Edelman
Mentor, A Siemens Business
46871 Bayside Parkway
Fremont, CA 94538

***Abstract*- This paper will educate the reader on how to use transaction recording for effective debug in a UVM based SystemVerilog design and test bench. Transaction recording has been available for years, with various APIs available in various states of usability. This paper will solidify the usage of the APIs and enable readers to build their own transaction based verification environments.**

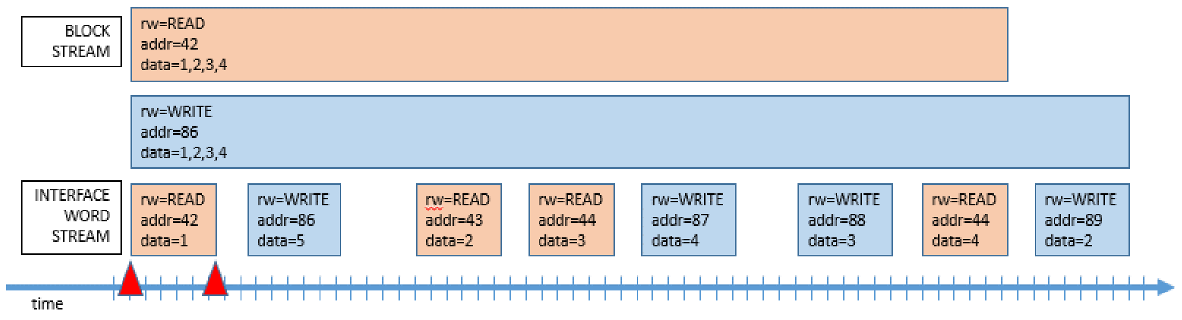
I. Introduction

Transaction recording seems at times to have a life of its own. It has existed since before simulation. Writing down the communication between two people – that's transaction recording.

In verification it isn't quite as easy as that. Transaction recording has been implemented by vendors and a UVM interface exists, but it is poorly implemented and hard to use. Many VIP providers use transaction recording to create better debug. Any user can use transaction recording to create better debug. The simple API below is very small, and very easy to use. The UVM API is a general API, but is poorly thought out. It is usable, but a better recommendation is to use a stand-alone kind of interface, as described in this paper. One upside of the UVM transaction recording is that it comes for free just by using sequences and sequence items. The free is limited to those areas. (Drivers and monitors are not instrumented automatically).

II. Transaction Recording – The Concepts

The concepts behind transaction recording are simple. A transaction has a beginning and an end. It can have attributes (properties). It can have relationships with other transaction (transaction A is a child of transaction B). A transaction lives on a "stream". A stream is a construct that enables transactions to be organized (my stream of AXI transactions), but also allows for easy reasoning and display. A stream is simply a horizontal line on the waveform display. On that horizontal line, transactions may be drawn.



Transaction Start and End

A transaction has a start time and an end time. They represent the time during which this transaction was active. A transaction start and end times can be the same time – a zero delay transaction. Start and End times can be in the future or in the past. In the diagram above, the RED triangles designate the start and end times for the READ transaction.

The diagram lays out transactions as they might be seen in the wave window. Time increases from left to right. Each "row" or horizontal display area is a stream.

Transaction Attributes

A transaction has attributes. Attributes are (NAME, VALUE) property pairs. For example a transaction could have the attributes "ADDR" and "DATA". Attributes have no special meaning to the transaction recording system or the simulation – they are useful to the verification or design engineer. They may be primary attributes like data and address, or a secondary (derived) attributes like "bytes per second". Attributes are very useful for debug and analysis.

In the diagram above, the transactions have attributes of 'rw', 'addr' and 'data'.

A Stream of Transactions

A transaction with a start time, end time and attributes lives on a stream. A stream is a virtual collection of transactions. A stream is simply a convenient place to think about where transactions exist. For example, a simulation could use just one stream for the entire simulation. That stream of transactions represents all the transactions of the simulation. Alternatively, EACH BUS could have a stream of transactions. In this case each stream of transactions represents all the transactions of the specific BUS. Instead, streams could represent activity on monitors and drivers. A UVM monitor could have a stream. The stream of transactions represents the transactions that passed through that monitor. Similarly for the driver. There could be a READ stream and a WRITE stream.

Relationships

Relationships between transactions can be useful for debug and analysis. For example, transaction X started transactions A, B and C. (X is the parent of A, B and C). Or transaction Y caused Z. Y is the predecessor of Z. Z is the successor of Y. Relations are normally quite hard to identify and capture. In addition, there are many other ways to understand how transactions are related which are more natural, and which survive going from UVM Testbench to SystemVerilog Interface to RTL to SystemVerilog Interface to UVM Testbench. For example an 'ID' field or transaction TAG. Or simply "the address".

III. TRANSACTION RECORDING – THE SIMPLE PLI CALLS

In the recent past a proposal for a simple PLI transaction recording API [1] was created. An API based loosely on this proposed standard is described below. This simple PLI API can be used to implement what was discussed in the previous section.

$create_transaction_stream

First a transaction stream must be created. In the UVM implementation, there is a "kind" field which can be ignored. It is supported in the PLI API, but not used. (In the UVM variously, the kind field may be "TVM", "Begin_No_Parent, Link" or "Begin_End, Link". These kind names exist in the implementation, but not in the UVM LRM).

```
HANDLE stream_handle = $create_transaction_stream (string stream_name, string kind);
```

$create_transaction_stream () is the start of transaction recording. It creates a "stream" named "stream_name" which will later "host" transactions. An example usage is

```
integer s;
s = $create_transaction_stream ("my_stream", "kind");
```

The "kind" argument is currently ignored.

$begin_transaction

Once a stream is available, transactions can be created on that stream. Use $begin_transaction to create a transaction.

```
HANDLE transaction_handle = $begin_transaction (HANDLE stream_handle, string name);
```

The stream handle returned from $create_transaction_stream () is the first argument. The second argument is the name of the transaction, for example "READ" or "WRITE" or "trans11234". This is an arbitrary name, and does not have to be unique.

$end_transaction

$free_transaction

A transaction is ended using the pair $end_transaction () and $free_transaction ().

```
$end_transaction (HANDLE transaction_handle)
$free_transaction (HANDLE transaction_handle)
```

A transaction that has had $end_transaction () called is ended, but is still available to be the source or target of relations. Some tools may decide to keep all the transactions available until the end of simulation, and then create relations between appropriate source and target transactions all at once, then exit simulation. This approach is ill advised, since there may be millions of transactions created during simulation, and they will consume memory. A better approach is to create relations as needed and then $free_transaction () the transactions. Although relations and transaction relationships could have a high value, the usual cost of creating them combined with their general lack of usefulness means that relations are rarely used and rarely recorded.

$add_relation

$add_relation creates a relationship of "relationship_name" between handle1 and handle2.

```
$add_relation (HANDLE source_transaction_handle1,
              HANDLE target_transaction_handle2, string relationship_name)
```

The UVM records relations in a way that is relatively unusable. The most useful relation a transaction has is the relation it has with any signals that it may control or monitor.

Relations are useful grouping techniques, but creating the database to support relations is hard. For example, a master might start a transaction to do a BLOCK read. It would need to manage and link any "children" transactions from that BLOCK (i.e. 16 byte READS, word READS, byte READS). Creating and maintaining the relationship database is beyond the scope of this paper. In the simplest case, relations can be managed with an "ID". Each related transaction has the same ID (or shares parts of the ID). Interestingly, using this technique leads to better relationship visualization and debug with simple search, no database is required, and no recording of links is required, and the relationships can be tracked across test bench and RTL. This is future work.

$add_attribute

With a stream and the ability to begin and end transaction and create relations, the $add_attribute API adds attributes to the transaction. Adding an attribute is like tagging it with data, or instrumenting it. The attributes describe "interesting properties" about the transaction.

For example, a READ transaction might have an ADDRESS attribute and a DATA attribute.

```
$add_attribute (HANDLE transaction_handle, OBJ value [, string attribute_name])
```

The $add_attribute API is very flexible. The "value" is a Verilog (simulator) object handle. It is "type-less". (It can be any type). The type is established in the simulator call, and recorded. There is no need for "type specific" attribute recording. These type-based variations are NOT needed: 'add_attribute_int ()', 'add_attribute_real ()', etc. Any type argument can be passed in to $add_attribute. The UVM recording API suffers from needing a type based interface.

$add_color

Color is a special attribute. It tells the display or debugger to paint this transaction in a particular color. The eye is quite good at finding color. Transactions which are colored can help significantly in debug.

```
$add_color (HANDLE transaction_handle, string color_name)
```

Color_name must be a known color name ("red") or an RGB value in the form "#RRGGBB".

IV. A Simple Example Usage

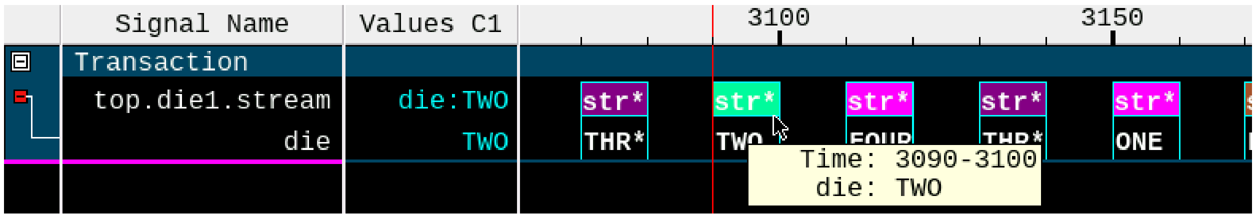
Given an enumeration from 1 to 6, and a Verilog module that calls $urandom_range to create "dice rolls", an example can be instrumented with transaction recording.

```
typedef enum bit[2:0] { NA, ONE, TWO, THREE, FOUR, FIVE, SIX } die_t;

module DIE(input clk);
  int stream, tr;
  die_t die;

  initial
    stream = $create_transaction_stream("stream", "kind");

  always @(posedge clk) begin
    die = die_t'($urandom_range(6,1));
    tr = $begin_transaction(stream, "roll");
    $add_attribute(tr, die.name(), "die");
    @(negedge clk);
    $end_transaction(tr);
    $free_transaction(tr);
    #1;
  end
endmodule
```



```
Time: 3090-3100
die: TWO
```

V. Transaction Recording – The UVM Layer

The UVM contains a transaction recording API. UVM-1.1d had a variety of API entry points and usages. UVM-1.2 and UVM IEEE added a few more and deprecated a few more. Frankly, both 1.2 and the IEEE versions are a bit of a mess. This provides great opportunity for improvements from the community. The built-in UVM transaction recording does provide enough functionality to be useful. It can be quite useful. However, using the entire UVM transaction recording API, and descending in many of the more esoteric data structures is not advised. The best advice is to keep it as simple as possible and stay away from the details.

In the next example, the UVM Transaction recording API can be analyzed and understood. The built-in recording is used, and additional do_record () implementation are provided. This is the simplest usage, and the recommended usage.

To record attributes in do_record, use the macro `uvm_record_field(). This macro provides the ability to use the underlying $add_attribute () in a native way, which will allow the data type to be created faithfully by the PLI and the simulator.

In the UVM, using the helpful macro, `uvm_record_field("name", get_name()), expands to

```
if (recorder != null && recorder.tr_handle != 0) begin
  if (recorder.get_type_name() != "uvm_recorder") begin
   $add_attribute(recorder.tr_handle, get_name(), "name");
  end
  else
    recorder.m_set_attribute(recorder.tr_handle,"name",$sformatf("%p",get_name()));
end
```

The RED line above is the desired attribute recording method. Using $sformatf and the “%p” model is fine, but for a user defined type like a struct, or a 13 bit bit-vector, the actual data type will get splattered into a string or an ‘int’, respectively.

Create a transaction class, and add do_record

```
class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)
  rw_t       rw;
  bit [31:0] addr;
  bit [31:0] data;
  rand int duration;

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field ("name", get_name ())
    `uvm_record_field ("rw", rw.name ())
    `uvm_record_field ("addr", addr)
    `uvm_record_field ("data", data)
    `uvm_record_field ("duration", duration)
  endfunction
endclass
```

Create a sequence class, creating and executing transactions.

```
class write_read_sequence extends uvm_sequence#(transaction);
  `uvm_object_utils(write_read_sequence)
  transaction tw;
  transaction tr;
  bit [31:0] start_addr;
  bit [31:0] end_addr;
  ...
  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("start_addr", start_addr)
    `uvm_record_field("end_addr", end_addr)
  endfunction

  task body();
    for (int i = start_addr; i < end_addr; i++) begin
      tw = transaction::type_id::create($sformatf("tw%0d", i));
      start_item(tw);
      if (!tw.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      tw.rw = WRITE;
      tw.addr = i ;
      tw.data = i+1 ;
      finish_item(tw);
      ...
    end
  endtask
endclass
```

In a test or other place, start a sequence

```
class test extends uvm_test;
  `uvm_component_utils(test)

  agent a1;
  write_read_sequence seq1;

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    seq1 = write_read_sequence::type_id::create("seq");
    seq1.mif = mif;
    seq1.start_addr = 1;
    seq1.end_addr   = 100;
    seq1.start(a1.sqr);
    ...
```

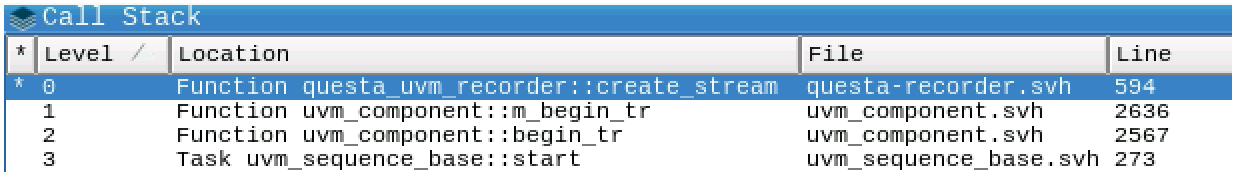
The RED seq1.start() is the entry point into the UVM transaction recording usage for starting a sequence.

The RED lines from above are the entry points into the UVM transaction recording usage for a sequence body.

```
start_item(tw);
finish_item(tw);
seq1.start(a1.sqr);
```

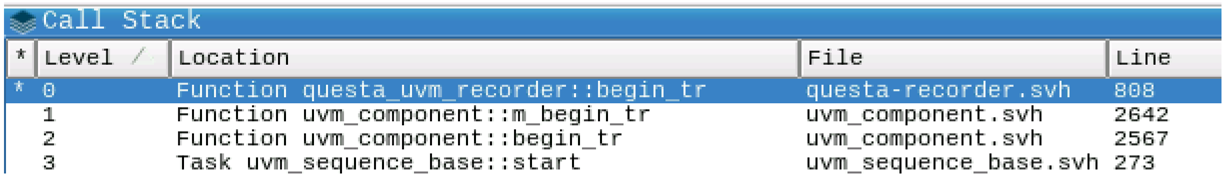
Start_item causes a "request" to be sent to the sequencer. Once the sequencer is ready to grant permission, then start_item will return and the finish_item is called to execute the actual transaction on the driver.

Calling seq1.start() will call uvm_recorder::create_stream().



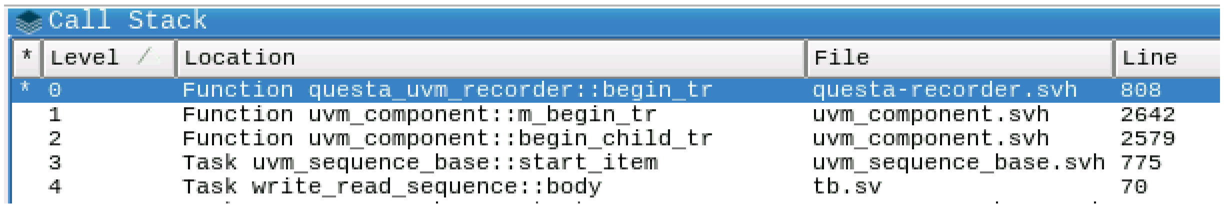
```
Call Stack
* | Level / | Location                                       | File                  | Line
* 0           Function questa_uvm_recorder::create_stream      questa-recorder.svh     594
  1           Function uvm_component::m_begin_tr               uvm_component.svh       2636
  2           Function uvm_component::begin_tr                 uvm_component.svh       2567
  3           Task uvm_sequence_base::start                    uvm_sequence_base.svh   273
```

Within that same seq1.start(), begin_tr will be called. This call starts the "sequence" recording.



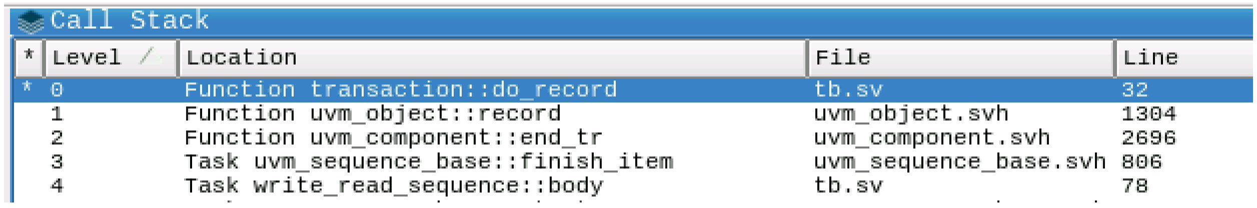
```
Call Stack
* | Level / | Location                                       | File                  | Line
* 0           Function questa_uvm_recorder::begin_tr           questa-recorder.svh     808
  1           Function uvm_component::m_begin_tr               uvm_component.svh       2642
  2           Function uvm_component::begin_tr                 uvm_component.svh       2567
  3           Task uvm_sequence_base::start                    uvm_sequence_base.svh   273
```

Calling start_item(tw) will call begin_tr as well. This call starts the "transaction" recording.



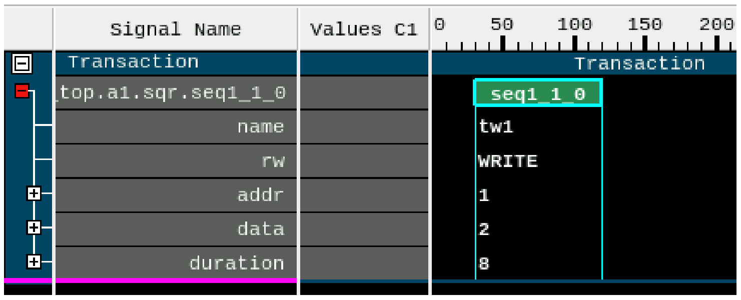
```
Call Stack
* | Level / | Location                                       | File                  | Line
* 0           Function questa_uvm_recorder::begin_tr           questa-recorder.svh     808
  1           Function uvm_component::m_begin_tr               uvm_component.svh       2642
  2           Function uvm_component::begin_child_tr           uvm_component.svh       2579
  3           Task uvm_sequence_base::start_item               uvm_sequence_base.svh   775
  4           Task write_read_sequence::body                   tb.sv                   70
```

Finally finish_item(tw) will cause end_tr to be called, which in turn will call do_record. The attributes are recorded at the END of a transaction.



```
Call Stack
* | Level / | Location                                       | File                  | Line
* 0           Function transaction::do_record                  tb.sv                   32
  1           Function uvm_object::record                      uvm_object.svh          1304
  2           Function uvm_component::end_tr                   uvm_component.svh       2696
  3           Task uvm_sequence_base::finish_item              uvm_sequence_base.svh   806
  4           Task write_read_sequence::body                   tb.sv                   78
```

Taken together these start(), start_item(), finish_item() and do_record() work together to produce a transaction that is viewable in the waveform or other debug tools.



You can also call begin_tr, end_tr, create_stream yourself. Using this level of instrumentation in your code is not advised. This is not a rich API, nor one designed for general use. The API has too many shortcomings to list. Using a live simulation debug session to single step through them will help with their limitations and usage. These API calls can be used outside the UVM built-in recording, but there are many assumptions on state within the UVM usage. It may be hard to debug and hard to use. There are about 75 API calls.

VI. USING BIND

The bind will effectively bind in a monitor to "listen" to the signals and wires that are of interest. It will determine what constitutes a "transaction" and determine when and how to record it using the SIMPLE recording API.

```
module dut_transaction_monitor(input CLK, input bit READY,
                                          input bit VALID,
       input rw_t rw,  input bit [31:0]addr,  input bit [31:0]rd,  input bit [31:0]wd);
  int s, tr;

  initial s = $create_transaction_stream("s", "kind");

  always @(posedge CLK) begin
    if ((READY==1) && (VALID==1)) begin
      tr = $begin_transaction(s, rw.name());
      $add_attribute(tr, rw.name(), "rw");
      $add_attribute(tr, addr, "addr");
      if (rw == READ) begin
        $add_attribute(tr, rd, "rd");
      end
      else if (rw == WRITE) begin
        $add_attribute(tr, wd, "wd");
      end
      while ((READY==1) && (VALID==1))
        @(posedge CLK);
      @(negedge CLK);
      $end_transaction(tr);
      $free_transaction(tr);
    end
  end
endmodule
```

Using the bind is easy. The bind file above is compiled, and then it is bound into the instances of interest. In this example, there is just one instances of interest - the simple 'dut'

In the example top module, the transaction monitor is bound in using 'bind':

```
module top();
  bind dut dut_transaction_monitor CHANNEL1(CLK, READY,  VALID,  rw,  addr,  rd,  wd);
  bind dut dut_transaction_monitor CHANNEL2(CLK, READY2, VALID2, rw2, addr2, rd2, wd2);
  …
endmodule
```

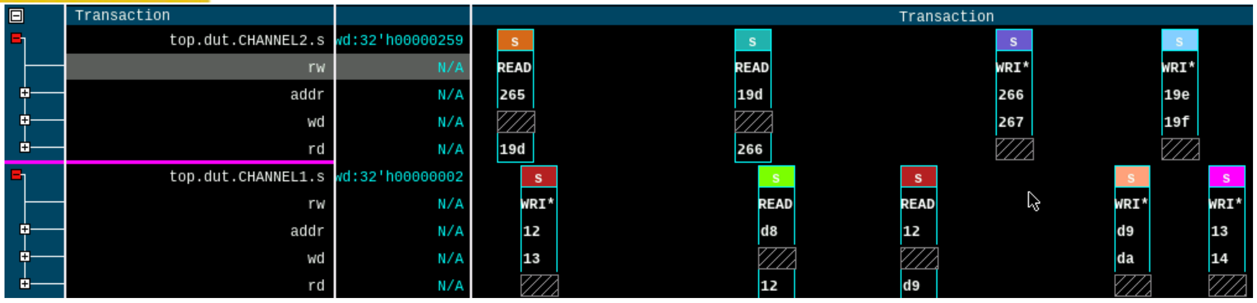
The dut

```
module my_dut(input CLK, output bit READY, input bit VALID,
    input rw_t rw,   input reg [31:0]addr,   output reg [31:0]rd,   input reg [31:0]wd,
                         output bit READY2, input bit VALID2,
    input rw_t rw2, input reg [31:0]addr2, output reg [31:0]rd2, input reg [31:0]wd2);

  bit [31:0] mem [1023:0] = '{default: 42};

  // READY control
  ...
  // Channel 1
  always @(posedge CLK) begin
    if ((READY == 1) && (VALID == 1)) begin
      if (rw == READ) begin
        rd = mem[addr];
      end
      else if (rw == WRITE) begin
        mem[addr] = wd;
      end
    end
    READY <= 0;
    @(negedge CLK);
  end

  // Channel 2
  always @(posedge CLK) begin
    if ((READY2 == 1) && (VALID2 == 1)) begin
      if (rw2 == READ) begin
        rd2 = mem[addr2];
      end
      else if (rw2 == WRITE) begin
        mem[addr2] = wd2;
      end
    end
    READY2 <= 0;
    @(negedge CLK);
  end
endmodule
```



VII. Conclusion

This paper has demonstrated a complete transaction recording system for both a stand-alone implementation and a UVM based implementation. Transaction debug can provide visibility where there previously was none. Even in the creation of the examples for this paper, transactions provided several “ah-ha” moments which explained the bug in the example code. It made debug much faster, and only required a small amount of work to implement.

The reader is left with the task of using the simple stand-alone recording API to instrument their own code. Any suggestions or improvements would be much appreciated. All source code is available from the author. Additional future work will investigate how transaction recording changed with UVM IEEE.

VIII. References

[1] UVM LRM, https://standards.ieee.org/standard/1800_2-2017.html
[2] SystemVerilog, 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language https://ieeexplore.ieee.org/document/8299595/citations#citations
[3] “Draft Standard for Verilog Transaction Recording Extensions”, http://www.boyd.com/1364_btf/report/full_pr/attach/435_IEEE_TR_Proposal_04.pdf

IX. Appendix – Complete DICE Example

```
int gid;

typedef enum bit[2:0] { NA, ONE, TWO, THREE, FOUR, FIVE, SIX } die_t;

module DIE(input clk);
  int stream;
  int tr;
  die_t die;

  initial
    stream = $create_transaction_stream("stream", "kind");

  always @(posedge clk) begin
    die = die_t'($urandom_range(6,1));
    tr = $begin_transaction(stream, "roll");
    $add_attribute(tr, die.name(), "die");
    @(negedge clk);
    $end_transaction(tr);
    #1;
  end
endmodule

module top();
  reg clk;

  DIE die1(clk);

  initial begin
    repeat (1000)
      @(posedge clk);
    $finish(2);
  end

  always begin
    clk = 0;
    #10;
    clk = 1;
    #10;
  end
endmodule
```

X. Appendix – Complete UVM Example

```
// =====================================================
//
//   File: t.sv
//
// =====================================================
import questa_uvm_pkg::*;

import uvm_pkg::*;
`include "uvm_macros.svh"

import types_pkg::*;

import tb_pkg::*;

interface my_if (input CLK );
  bit READY;
  bit VALID;

  rw_t rw;
  reg [31:0]addr;
  reg [31:0]rd;
  reg [31:0]wd;
endinterface


module top();
  reg CLK;

  bind dut dut_transaction_monitor CHANNEL1(CLK, READY,  VALID,  rw,  addr,  rd,  wd);
  bind dut dut_transaction_monitor CHANNEL2(CLK, READY2, VALID2, rw2, addr2, rd2, wd2);

  my_if   mif(CLK);
  my_if  mif2(CLK);

  my_dut dut(CLK,
     mif.READY,  mif.VALID,  mif.rw,  mif.addr,  mif.rd,  mif.wd,
    mif2.READY, mif2.VALID, mif2.rw, mif2.addr, mif2.rd, mif2.wd);

  initial begin
    uvm_config_db#(virtual my_if)::set(null, "*", "mif",   mif);
    uvm_config_db#(virtual my_if)::set(null, "*", "mif2", mif2);

    run_test();
  end

  always begin
    #10 CLK = 1;
    #10 CLK = 0;
  end
endmodule
```

```
// ======================================================
//
//   File: tb.sv
//
// ======================================================
package tb_pkg;

import uvm_pkg::*;
`include "uvm_macros.svh"

import types_pkg::*;

class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

  rw_t       rw;
  bit [31:0] addr;
  bit [31:0] data;

  rand int duration;

  constraint value {
    duration > 3;
    duration < 10;
  };

  function new(string name = "transaction");
    super.new(name);
  endfunction

  function string convert2string();
    return $sformatf("[%s] rw=%s, addr=%0d, data=%0d (%0d)", get_type_name(), rw.name(),
addr, data, duration);
  endfunction

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("rw", rw.name())
    `uvm_record_field("addr", addr)
    `uvm_record_field("data", data)
    `uvm_record_field("duration", duration)
  endfunction
endclass

class write_read_sequence extends uvm_sequence#(transaction);
  `uvm_object_utils(write_read_sequence)

  virtual my_if mif;
```

```
int delay;

transaction tw;
transaction tr;
bit [31:0] start_addr;
bit [31:0] end_addr;

function new(string name = "write_read_sequence");
  super.new(name);
endfunction

function void do_record(uvm_recorder recorder);
  super.do_record(recorder);
  `uvm_record_field("name", get_name())
  `uvm_record_field("start_addr", start_addr)
  `uvm_record_field("end_addr", end_addr)
endfunction

task body();
  `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)

  for (int i = start_addr; i < end_addr; i++) begin
    delay = $urandom_range(10, 2);
    repeat(delay) @(posedge mif.CLK);
    tw = transaction::type_id::create($sformatf("tw%0d", i));
    start_item(tw);
    if (!tw.randomize())
      `uvm_fatal(get_type_name(), "Randomize FAILED")
    tw.rw = WRITE;
    tw.addr = i ;
    tw.data = i+1 ;
    delay = $urandom_range(3, 2);
    repeat(delay) @(posedge mif.CLK);
    finish_item(tw);

    delay = $urandom_range(10, 2);
    repeat(delay) @(posedge mif.CLK);
    tr = transaction::type_id::create($sformatf("tr%0d", i));
    start_item(tr);
    if (!tr.randomize())
      `uvm_fatal(get_type_name(), "Randomize FAILED")
    tr.rw = READ;
    tr.addr = i ;
    tr.data = i+1 ;
    delay = $urandom_range(3, 2);
    repeat(delay) @(posedge mif.CLK);
    finish_item(tr);

    // Check
    if (tr.data != tw.data) begin
```

```
        `uvm_info(get_type_name(), $sformatf("ERROR: wrote '%0d', read '%0d'",
          tw.data, tr.data), UVM_MEDIUM)
        `uvm_fatal(get_type_name(), "MISMATCH")
      end
      else begin
        `uvm_info(get_type_name(), $sformatf("MATCH: wrote '%0d', read '%0d'",
          tw.data, tr.data), UVM_MEDIUM)
      end
    end
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass

class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

  virtual my_if mif;
  transaction t;

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

  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);
      `uvm_info(get_type_name(), $sformatf("Got %s", t.convert2string()), UVM_MEDIUM)
      #(t.duration);
      if (t.rw == READ) begin
        mif.rw = t.rw;
        mif.addr = t.addr;
        mif.VALID = 1;
        @(posedge mif.CLK);
        forever begin
          if (mif.READY == 1) break;
          @(posedge mif.CLK);
        end
        @(posedge mif.CLK);
        t.data = mif.rd;
        @(negedge mif.CLK);
        mif.VALID = 0;
        mif.rd = 'z;
        @(negedge mif.CLK);
      end
      else if (t.rw == WRITE) begin
        mif.rw = t.rw;
        mif.addr = t.addr;
        mif.wd = t.data;
        mif.VALID = 1;
        @(posedge mif.CLK);
        forever begin
```

```
          if (mif.READY == 1) break;
          @(posedge mif.CLK);
        end
        mif.VALID = 0;
        mif.wd = 'z;
        @(negedge mif.CLK);
      end
      seq_item_port.item_done();
    end
  endtask
endclass

class agent extends uvm_agent;
  `uvm_component_utils(agent)

  uvm_sequencer#(transaction) sqr;
  driver d;

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

  function void build_phase(uvm_phase phase);
    sqr = uvm_sequencer#(transaction)::type_id::create("sqr", this);
    d = driver::type_id::create("d", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    d.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass

class test extends uvm_test;
  `uvm_component_utils(test)

  virtual my_if mif;
  virtual my_if mif2;

  agent a1;
  agent a2;

  write_read_sequence seq1_1[4];
  write_read_sequence seq2_1[4];
  write_read_sequence seq1_2[4];
  write_read_sequence seq2_2[4];

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

  function void build_phase(uvm_phase phase);
```

```
    if (!uvm_config_db#(virtual my_if)::get( this, "", "mif", mif)) begin
      `uvm_info(get_type_name(), "GET CONFIG mif FAILED", UVM_MEDIUM)
      `uvm_fatal(get_type_name(), "CONFIG LOOKUP FAILED")
    end
    else
      `uvm_info(get_type_name(), "GET CONFIG mif OK", UVM_MEDIUM)

    if (!uvm_config_db#(virtual my_if)::get( this, "", "mif2", mif2)) begin
      `uvm_info(get_type_name(), "GET CONFIG mif2 FAILED", UVM_MEDIUM)
      `uvm_fatal(get_type_name(), "CONFIG LOOKUP FAILED")
    end
    else
      `uvm_info(get_type_name(), "GET CONFIG mif2 OK", UVM_MEDIUM)

    a1 = agent::type_id::create("a1", this);
    a2 = agent::type_id::create("a2", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    a1.d.mif = mif;
    a2.d.mif = mif2;
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    for (int i = 0; i < 1; i++) begin
      fork
        automatic int j = i;
        begin
          seq1_1[j] = write_read_sequence::type_id::create($sformatf("seq1_1-%0d", j));
          seq1_1[j].mif = mif;
          seq1_1[j].start_addr = 1;
          seq1_1[j].end_addr   = 100;
          seq1_1[j].start(a1.sqr);
        end
      join_none
      fork
        automatic int j = i;
        begin
          seq2_1[j] = write_read_sequence::type_id::create($sformatf("seq2_1-%0d", j));
          seq2_1[j].mif = mif;
          seq2_1[j].start_addr = 200;
          seq2_1[j].end_addr   = 300;
          seq2_1[j].start(a1.sqr);
        end
      join_none
      fork
        automatic int j = i;
        begin
          seq1_2[j] = write_read_sequence::type_id::create($sformatf("seq1_2-%0d", j));
          seq1_2[j].mif = mif2;
          seq1_2[j].start_addr = 400;
          seq1_2[j].end_addr   = 500;
          seq1_2[j].start(a2.sqr);
        end
      join_none
      fork
        automatic int j = i;
        begin
          seq2_2[j] = write_read_sequence::type_id::create($sformatf("seq2_2-%0d", j));
          seq2_2[j].mif = mif2;
          seq2_2[j].start_addr = 600;
          seq2_2[j].end_addr   = 700;
          seq2_2[j].start(a2.sqr);
        end
      join_none
    end
    wait fork;
    phase.drop_objection(this);
  endtask
endclass

endpackage


// =====================================================
//
//   File: bind.sv
//
// =====================================================
import types_pkg::*;

module dut_transaction_monitor(input CLK, input bit READY,
                                          input bit VALID,
    input rw_t rw,  input bit [31:0]addr,  input bit [31:0]rd,  input bit [31:0]wd);
  int s;
  int tr;

  initial
    s = $create_transaction_stream("s", "kind");

  always @(posedge CLK) begin
    if ((READY==1) && (VALID==1)) begin
      tr = $begin_transaction(s, rw.name());
      $add_attribute(tr, rw.name(), "rw");
      $add_attribute(tr, addr, "addr");
      if (rw == READ) begin
        $add_attribute(tr, rd, "rd");
      end
      else if (rw == WRITE) begin
        $add_attribute(tr, wd, "wd");
      end
      while ((READY==1) && (VALID==1))
        @(posedge CLK);
      @(negedge CLK);
      $end_transaction(tr);
      $free_transaction(tr);
    end
  end
endmodule


// =====================================================
//
//   File: dut.sv
//
// =====================================================
import types_pkg::*;

module my_dut(input CLK, output bit READY,  input bit VALID,
       input rw_t rw,  input reg [31:0]addr,  output reg [31:0]rd,  input reg [31:0]wd,
                         output bit READY2, input bit VALID2,
       input rw_t rw2, input reg [31:0]addr2, output reg [31:0]rd2, input reg [31:0]wd2);

  bit [31:0] mem[1023:0] = '{default: 42};

  always begin
    int d;
    READY <= 1;
    @(posedge CLK);
    d = $urandom_range(5, 3);
    repeat(d) @(posedge CLK);
  end

  always begin
    int d;
    READY2 <= 1;
    @(posedge CLK);
    d = $urandom_range(10, 6);
    repeat(d) @(posedge CLK);
  end

  always @(posedge CLK) begin
    if ((READY == 1) && (VALID == 1)) begin
      if (rw == READ) begin
        rd = mem[addr];
      end
      else if (rw == WRITE) begin
        mem[addr] = wd;
      end
    end
    READY <= 0;
    @(negedge CLK);
  end

  always @(posedge CLK) begin
    if ((READY2 == 1) && (VALID2 == 1)) begin
      if (rw2 == READ) begin
        rd2 = mem[addr2];
      end
      else if (rw2 == WRITE) begin
        mem[addr2] = wd2;
      end
    end
    READY2 <= 0;
    @(negedge CLK);
  end
endmodule


// =====================================================
//
//   File: types.sv
//
// =====================================================
package types_pkg;
  typedef enum bit[1:0] {READ, WRITE, IDLE} rw_t;
endpackage
```