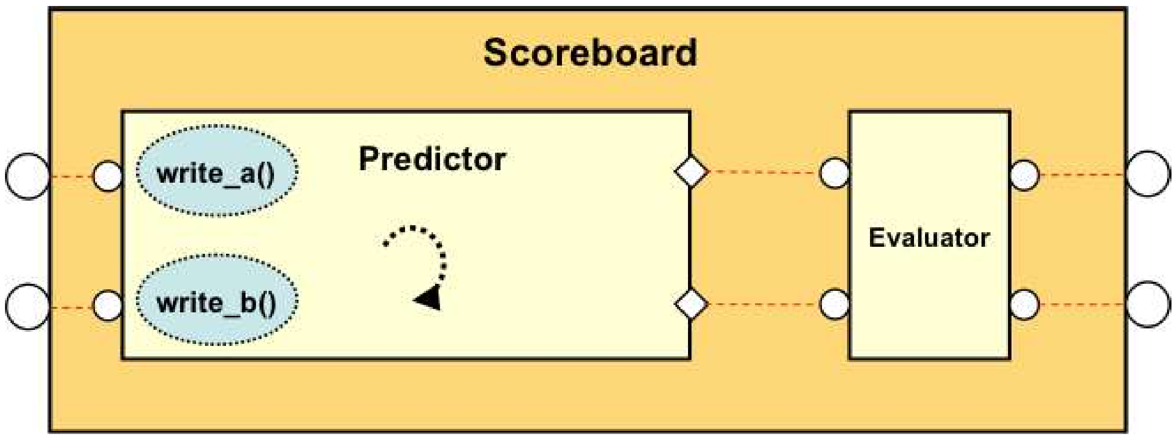
Predictors

Overview

A Predictor is a verification component that represents a "golden" reference model of all or part of the DUT functionality. It takes the same input stimulus that is sent to the DUT and produces expected response data that is by definition correct. When you send random stimulus into your DUT, you need an automatic way to check the result as you can no longer check the output by hand. A Predictor generates expected output that is compared to the actual DUT output to give a pass / fail.

Predictors in the Testbench Environment



Predictors are the part of the Scoreboard component that generates expected transactions. They should be separate from the part of the Scoreboard that performs the evaluation. A Predictor can have one or more input streams, which are the same input streams that are applied to the DUT.

Construction

Predictors are typical analysis components that are subscribers to transaction streams. The inputs to a Predictor are transactions generated from monitors observing the input interfaces of the DUT. The Predictors take the input transaction(s) and process them to produce expected output transactions. Those output transactions are broadcast through analysis ports to the evaluator part of the scoreboard, and to any other analysis component that needs to observe predicted transactions. Internally, Predictors can be written in C, C++, SV or SystemC, and are written at an abstract level of modeling. Since Predictors are written at the transaction level, they can be readily chained if needed.

Example

```
class alu_tlm extends uvm_subscriber #(alu_txn);
 `uvm_component_utils(alu_tlm)

 uvm_analysis_port #(alu_txn) results_ap;

 function new(string name, uvm_component parent );
  super.new( name , parent );
 endfunction

 function void build_phase( uvm_phase phase );
```

```
  results_ap = new("results_ap", this);
 endfunction

 function void write( alu_txn t);
  alu_txn out_txn;
  $cast(out_txn,t.clone());
  case(t.mode)
   ADD: out_txn.result = t.val1 + t.val2;
   SUB: out_txn.result = t.val1 - t.val2;
   MUL: out_txn.result = t.val1 * t.val2;
   DIV: out_txn.result = t.val1 / t.val2;
  endcase
  results_ap.write(out_txn);
 endfunction

endclass
```

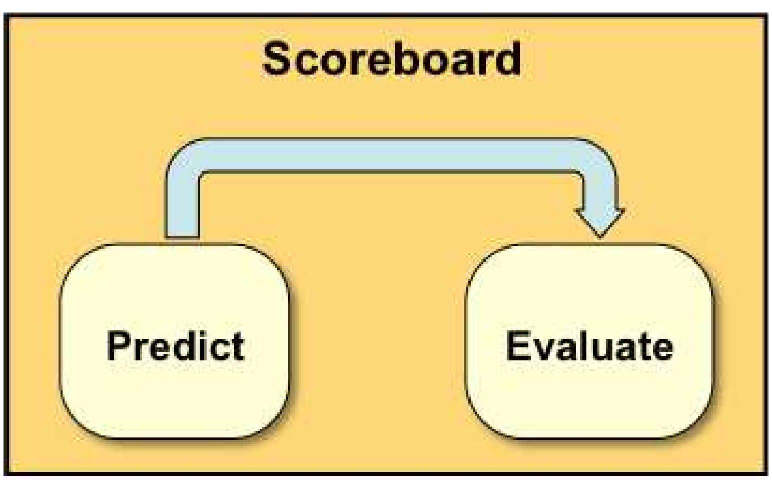
Predictor as a Proxy for the DUT

Another use of a Predictor is to act as a proxy DUT while the DUT is being written. Typically, since the Predictor is written at a higher level of abstraction, it takes less time to write and is available earlier than the DUT. As a proxy for the DUT, it allows testbench development and debugging to proceed in parallel with DUT development.

Scoreboards

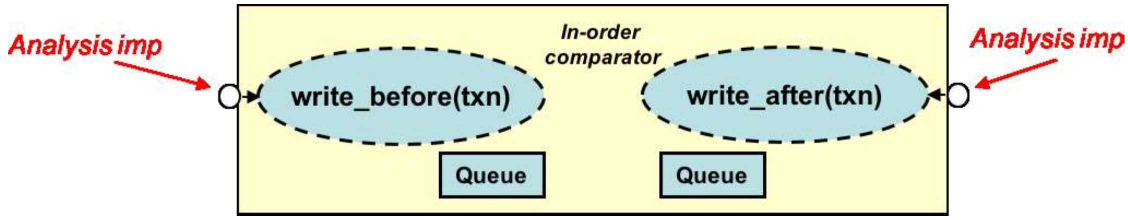
Overview

The Scoreboard's job is to determine whether or not the DUT is functioning properly. The scoreboard is usually the most difficult part of the testbench to write, but it can be generalized into two parts: The first step is determining what exactly is the correct functionality. Once the correct functionality is predicted, the scoreboard can then evaluate whether or not the actual results observed on the DUT match the predicted results. The best scoreboard architecture is to separate the prediction task from the evaluation task. This gives you the most flexibility for reuse by allowing for substitution of predictor and evaluation models, and follows the best practice of separation of concerns.



In cases where there is a single stream of predicted transactions and a single stream of actual transactions, the scoreboard can perform the evaluation with a simple comparator. The most common comparators are an in-order and out-of-order comparator.

Comparing Transactions Assuming In-Order Arrival

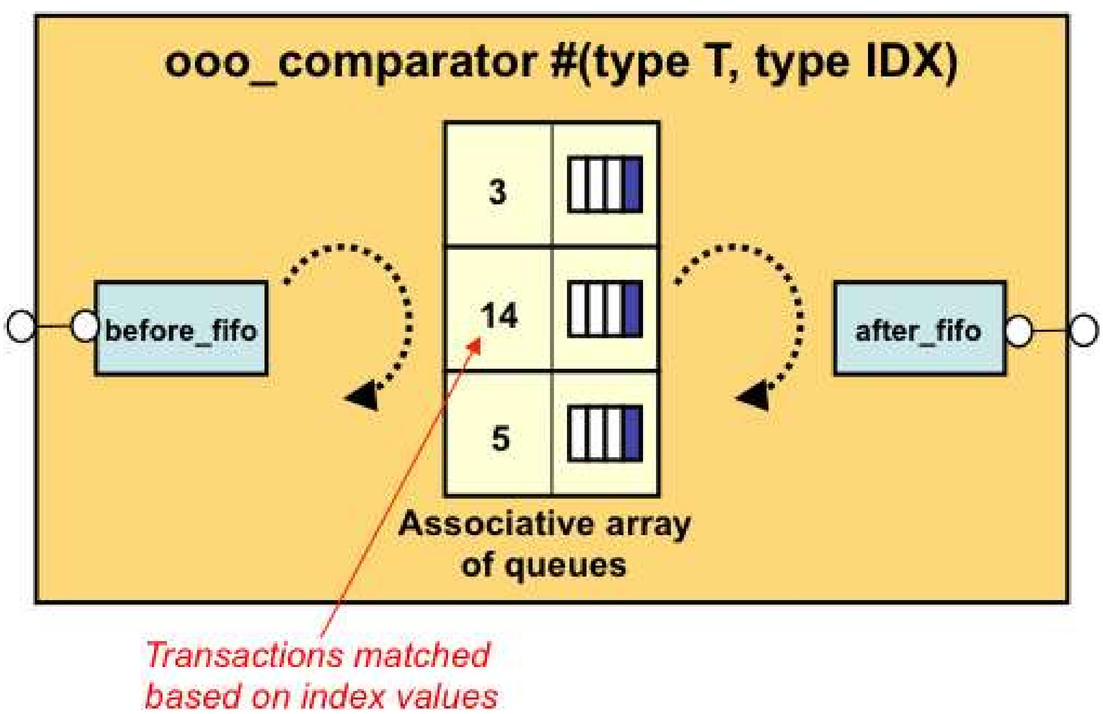


An in-order comparator assumes that matching transactions will appear in the same order from both expected and actual streams. It gets transactions from the expected and actual side and evaluates them. The transactions will arrive independently, but in order so transactions can be stored from the "before" side and then compared when a transaction arrives on the "after" side. Evaluation can be as simple as calling the transaction's compare() method, or it can be more involved, because for the purposes of evaluating correct behavior, comparison does not necessarily mean equality.

```
`uvm_analysis_imp_decl(_before)
`uvm_analysis_imp_decl(_after)

class comparator_inorder extends uvm_component;

  `uvm_component_utils(comparator_inorder)

  uvm_analysis_imp_before #(T, comparator_inorder) before_export;
  uvm_analysis_imp_after #(T, comparator_inorder) after_export;

  int m_matches, m_mismatches;
  protected T m_before[$];
  protected T m_after[$];

 function new( string name , uvm_component parent) ;
  super.new( name , parent );
  m_matches = 0;
  m_mismatches = 0;
 endfunction

 function void build_phase( uvm_phase phase );
   before_export = new("before_export", this);
   after_export = new("after_export", this);
 endfunction

 protected virtual function void m_proc_data();
   T bef = m_before.pop_front();
   T aft = m_after.pop_front();
   if (!bef.compare(aft)) begin
     `uvm_error("Comparator Mismatch",
         $sformatf("%s does not match %s",
                bef.convert2string(),
                aft.convert2string()))
     m_mismatches++;
```

```
   end
   else begin
     m_matches++;
   end
 endfunction

 virtual function void write_before(T txn);
    m_before.push_back(txn);
    if (m_after.size())
      m_proc_data();
 endfunction

 virtual function void write_after(T txn);
    m_after.push_back(txn);
    if (m_before.size())
      m_proc_data();
 endfunction

 function void report_phase( uvm_phase phase );
  `uvm_info("Inorder Comparator", $sformatf("Matches:    %0d",
m_matches), UVM_LOW);
  `uvm_info("Inorder Comparator", $sformatf("Mismatches: %0d",
m_mismatches), UVM_LOW);
 endfunction

endclass
```

Comparing transactions out-of-order



An out-of-order comparator makes no assumption that matching transactions will appear in the same order from the expected and actual sides. So, unmatched transactions need to be stored until a matching transaction appears on the opposite stream. For most out-of-order comparators, an associative array is used for storage. This example comparator has two input streams arriving through analysis exports. The implementation of the comparator is symmetrical, so the export names do not have any real importance. This example uses embedded fifos to implement the analysis write() functions, but since the transactions are either stored into the associative array or evaluated upon arrival, this example could easily be written using analysis imps and write() functions.

Because of the need to determine if two transactions are a match and should be compared, this example requires transactions to implement an index_id() function that returns a value that is used as a key for the associative array. If an entry with this key already exists in the associative array, it means that a transaction previously arrived from the other stream, and the transactions are compared. If no key exists, then this transaction is added to associative array.

This example has an additional feature in that it does not assume that the index_id() values are always unique on a given stream. In the case where multiple outstanding transactions from the same stream have the same index value, they are stored in a queue, and the queue is the value portion of the associative array. When matches from the other stream arrive, they are compared in FIFO order.

```
class ooo_comparator
  #(type T = int,
    type IDX = int)
   extends uvm_component;

  typedef ooo_comparator #(T, IDX) this_type;
  `uvm_component_param_utils(this_type)

  typedef T q_of_T[$];
  typedef IDX q_of_IDX[$];

  uvm_analysis_export #(T) before_axp, after_axp;

  protected uvm_tlm_analysis_fifo #(T) before_fifo, after_fifo;
  bit before_queued = 0;
  bit after_queued = 0;

  protected int m_matches, m_mismatches;

  protected q_of_T received_data[IDX];
  protected int rcv_count[IDX];

  protected process before_proc = null;
  protected process after_proc  = null;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase( uvm_phase phase );
    before_axp = new("before_axp", this);
```

```
        after_axp = new("after_axp", this); before_fifo =
        new("before", this); after_fifo = new("after", this);
    endfunction

    function void connect_phase( uvm_phase phase );
        before_axp.connect(before_fifo.analysis_export);
        after_axp.connect(after_fifo.analysis_export);
    endfunction : connect

// The component forks two concurrent instantiations of this task
// Each instantiation monitors an input analysis fifo
    protected task get_data(ref uvm_tlm_analysis_fifo #(T) txn_fifo,
input bit is_before);
        T txn_data, txn_existing; IDX idx;
        string rs; q_of_T
        tmpq;
        bit need_to_compare;
        forever begin

    // Get the transaction object, block if no transaction available
            txn_fifo.get(txn_data); idx =
            txn_data.index_id();

    // Check to see if there is an existing object to compare
            need_to_compare = (rcv_count.exists(idx) &&
                                        ((is_before && rcv_count[idx] > 0) || (!is_before &&
                                        rcv_count[idx] < 0)));
            if (need_to_compare) begin
    // Compare objects using compare() method of transaction
                tmpq = received_data[idx]; txn_existing =
                tmpq.pop_front(); received_data[idx] = tmpq;
                if (txn_data.compare(txn_existing)) m_matches++;
                else
                    m_mismatches++;
            end
            else begin
            // If no compare happened, add the new entry
                if (received_data.exists(idx)) tmpq =
                    received_data[idx];
                else
                    tmpq = {};
                tmpq.push_back(txn_data);
                received_data[idx] = tmpq;
```

```
      end

   // Update the index count
      if (is_before)
         if (rcv_count.exists(idx)) begin
            rcv_count[idx]--;
         end
         else
            rcv_count[idx] = -1;
      else
         if (rcv_count.exists(idx)) begin
            rcv_count[idx]++;
         end
         else
            rcv_count[idx] = 1;

   // If index count is balanced, remove entry from the arrays
      if (rcv_count[idx] == 0) begin
         received_data.delete(idx);
         rcv_count.delete(idx);
      end
   end // forever
endtask

virtual function int get_matches();
   return m_matches;
endfunction : get_matches

virtual function int get_mismatches();
   return m_mismatches;
endfunction : get_mismatches

virtual function int get_total_missing();
   int num_missing;
   foreach (rcv_count[i]) begin
      num_missing += (rcv_count[i] < 0 ? -rcv_count[i] : rcv_count[i]);
   end
   return num_missing;
endfunction : get_total_missing

virtual function q_of_IDX get_missing_indexes(); q_of_IDX rv =
   rcv_count.find_index() with (item != 0); return rv;
endfunction : get_missing_indexes;

virtual function int get_missing_index_count(IDX i);
// If count is < 0, more "before" txns were received
```

```
  // If count is > 0, more "after" txns were received
    if (rcv_count.exists(i))
      return rcv_count[i];
    else
      return 0;
  endfunction : get_missing_index_count;

  task run_phase( uvm_phase phase );
    fork
      get_data(before_fifo, before_proc, 1);
      get_data(after_fifo, after_proc, 0);
    join
  endtask : run_phase

endclass : ooo_comparator
```

Advanced Scenarios

In more advanced scenarios, there can be multiple predicted and actual transaction streams coming from multiple DUT interfaces. In this case, a simple comparator is insufficient and the implementation of the evaluation portion of the scoreboard is more complex and DUT-specific.

Reporting and Recording

The result of the evaluation is a boolean value, which the Scoreboard should use to report and record failures.
Usually successful evaluations are not individually reported, but can be recorded for later summary reports.

Metric Analyzers

Overview

Metric Analyzers watch and record non-functional behavior such as latency, power utilization, and other performance-related measurements.

Construction

Metric Analyzers are generally standard analysis components. They implement their behavior in a way that depends on the number of transaction streams they observe - either by extending uvm_subscriber or with analysis imp/exports. They can perform ongoing calculations during the run phase, and/or during the post-run phases.

Example

```
`uvm_analysis_imp_decl(_BEFORE)
`uvm_analysis_imp_decl(_AFTER)

class delay_analyzer extends uvm_component;
`uvm_component_utils(delay_analyzer)

  uvm_analysis_imp_BEFORE #(alu_txn, delay_analyzer) before_export;
  uvm_analysis_imp_AFTER  #(alu_txn, delay_analyzer) after_export;

  real m_before[$];
  real m_after[$];
  real last_b_time, last_a_time;
  real longest_b_delay, longest_a_delay;

 function new( string name , uvm_component parent) ;
  super.new( name , parent );
  last_b_time = 0.0;
  last_a_time = 0.0;
 endfunction

// Record when the transaction arrives
 function void write_BEFORE(alu_txn t);
   real delay;
   delay = $realtime - last_b_time;
   last_b_time = $realtime;
   m_before.push_back(delay);
 endfunction

// Record when the transaction arrives
 function void write_AFTER(alu_txn t);
   real delay;
   delay = $realtime - last_a_time;
   last_a_time = $realtime;
   m_after.push_back(delay);
```

```
 endfunction

 function void build_phase( uvm_phase phase );
   before_export = new("before_export", this);
   after_export = new("after_export", this);
 endfunction

// Perform calculation for longest delay metric
 function void extract_phase( uvm_phase phase );
   foreach (m_before[i])
     if (m_before[i] > longest_b_delay) longest_b_delay = m_before[i];

   foreach (m_after[i])
     if (m_after[i] > longest_a_delay) longest_a_delay = m_after[i];
 endfunction

 function void check_phase( uvm_phase phase );
   string s;
   if (longest_a_delay > 100.0) begin
     $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
     `uvm_warning("Delay Analyzer",s);
   end
   if (longest_b_delay > 100.0) begin
     $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
     `uvm_warning("Delay Analyzer",s);
   end
 endfunction

 function void report_phase( uvm_phase phase );
  `uvm_info("Delay Analyzer", $sformatf("Longest BEFORE delay:
%5.2f", longest_b_delay), UVM_LOW);
  `uvm_info("Delay Analyzer", $sformatf("Longest AFTER delay:
%5.2f", longest_a_delay), UVM_LOW);
 endfunction

endclass
```

Post-Run Phases

Overview

Many analysis components perform their analysis on an ongoing basis during the simulation run. Sometimes you need to defer analysis until all data has been collected, or a component might need to do a final check at the end of simulation. For these components, UVM provides the post-run phases extract, check, and report.
These phases are exected in a hierarchically bottom-up fashion on all components.

The Extract Phase

The extract phase allows a component to examine data collected during the simulation run, extract meaningful values and perform arithmetic computation on those values.

The Check Phase

The check phase allows a component to evaluate any values computed during the extract phase and make a judgment about whether the values are correct. Also, for components that perform analysis continuously during the run, the check phase can be used to check for any missing data or extra data such as unmatched transactions in a scoreboard.

The Report Phase

The report phase allows a component to display a final report about the analysis in its area of responsibility. A component can be configured whether or not to display its local results, to allow for accumulation and display by higher-level components.

The Final Phase

The Final phase is the very last UVM phase to be executed before the UVM executes $finish to bring the simulation to an end.

Example

```
`uvm_analysis_imp_decl(_BEFORE)
`uvm_analysis_imp_decl(_AFTER)

class delay_analyzer extends uvm_component;
`uvm_component_utils(delay_analyzer)

  uvm_analysis_imp_BEFORE #(alu_txn, delay_analyzer) before_export;
  uvm_analysis_imp_AFTER  #(alu_txn, delay_analyzer) after_export;

  real m_before[$];
  real m_after[$];
  real last_b_time, last_a_time;
  real longest_b_delay, longest_a_delay;

 function new( string name , uvm_component parent) ;
  super.new( name , parent );
  last_b_time = 0.0;
  last_a_time = 0.0;
```

```
endfunction

function void write_BEFORE(alu_txn t);
    real delay;
    delay = $realtime - last_b_time; last_b_time =
    $realtime; m_before.push_back(delay);
endfunction


function void write_AFTER(alu_txn t);
    real delay;
    delay = $realtime - last_a_time; last_a_time =
    $realtime; m_after.push_back(delay);
endfunction

function void build_phase( uvm_phase phase ); before_export =
    new("before_export", this); after_export = new("after_export",
    this);
endfunction

function void extract_phase( uvm_phase phase );
    foreach (m_before[i])
        if (m_before[i] > longest_b_delay) longest_b_delay = m_before[i];

    foreach (m_after[i])
        if (m_after[i] > longest_a_delay) longest_a_delay = m_after[i];
endfunction

function void check_phase( uvm_phase phase );
    string s;
    if (longest_a_delay > 100.0) begin
        $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
        `uvm_warning("Delay Analyzer",s);
    end
    if (longest_b_delay > 100.0) begin
        $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
        `uvm_warning("Delay Analyzer",s);
    end
endfunction

function void report_phase( uvm_phase phase );
   `uvm_info("Delay Analyzer", $sformatf("Longest BEFORE delay:
%5.2f", longest_b_delay), UVM_LOW);
   `uvm_info("Delay Analyzer", $sformatf("Longest AFTER delay:
%5.2f", longest_a_delay), UVM_LOW);
```

```
  endfunction

  function void final_phase( uvm_phase phase );
   my_summarize_test_results();
  endfunction

endclass
```