

# Versatile UVM Scoreboarding

Jacob Andersen, Peter Jensen, Kevin Steffensen, Kasper Hesse

SyoSil ApS, Høje-Taastrup, Denmark

{jacob, peter, kevin, kasper}@syosil.com

**Abstract** —UVM engineers employ scoreboarding for checking DUT/REF behavior, but only few spend their time wisely by employing an existing scoreboard architecture. The main reason for this is that existing frameworks have inadequately served user needs and offer limited debugging capabilities. This paper presents a better UVM scoreboard framework, focusing on scalability, architectural separation and connectivity to foreign environments. The proposed scoreboard architecture has successfully been used in UVM testbenches at various architectural levels, across models (RTL, SystemC) and on physical devices (FPGA/ASIC). Based on our work, the SV/UVM user ecosystem will be able to improve how scoreboards are designed, configured and reused across projects, applications and models/architectural levels.

**Keywords** — *SystemVerilog; UVM; Scoreboard Architecture; RTL; SystemC; TLM; Debug; OOP Design Patterns*

## I. MOTIVATION; EXISTING WORK

Addressing the increasing challenges met when performing functional verification, UVM proposes a firm and productive approach for how to build and reuse verification components, environments and sequences/tests. When it comes to describing how to scoreboard and check the behavior of your design against one or more reference models, UVM offers less help. UVM does not present a scoreboard architecture but leaves the implementer to extend the empty `uvm_scoreboard` base class into a custom scoreboard that connects to analysis ports. Experience shows that custom scoreboard implementations across different application domains have several common denominators of deficiency. Users struggle to implement checker mechanisms for the designs under test (DUT) being exposed to random stimuli without sacrificing aspects like efficiency, easy debug and a clean implementation.

Existing user donated work [1] presents some UVM scoreboard architectures, offering UVM base classes and implementation guidelines together with some examples of use. These implementations commonly require the user to write an “expect” function that can check the DUT’s responses once these transactions arrive at the scoreboard. The use of such a non-blocking function imposes numerous modeling restrictions. Most notably, only one model can be checked, namely the DUT. Secondly, it is difficult to model many of the parallel aspects of common DUT types, leading to unclean implementations where the comparison mechanism becomes interwoven with the queue modeling. The “expect” function may be suitable for rather simple applications (e.g. packet switching), but does not fit more generic use cases. Lastly, self-contained SystemC/TLM virtual prototypes are difficult to incorporate as “expect” functions in such scoreboard architectures.

We find that existing proposals do not adequately address our requirements for a state-of-the-art, scalable scoreboard architecture. Therefore, we have created a scoreboard implementation that has been used across multiple UVM verification projects, and we would like to share our experiences and the guidelines we have set up. For the UVM community to benefit from our work, our scoreboard library has been released and is available for download (see Section IX).

A version of this paper was previously published at DVCon 2014 and 2015 [2]. This version of the paper reiterates some of the contributions of the SyoSil Scoreboard, and introduces some major improvements to the scoreboard that have been implemented between 2019 and 2022.

## II. SCOREBOARD STRUCTURE

The scoreboard is designed with an emphasis on scalability and ease of configuration. For that reason, the scoreboard can handle any number of concurrent models that must be checked, as well as multiple comparison strategies, depending on the desired behavior. Figure 1 presents a generic UVM scoreboarding setup where the reference model (REF) and scoreboard are combined, and Figure 2 presents our proposed setup where the REF and scoreboard are separated. While it may seem intuitive to combine REF and scoreboard, doing so increases the complexity of both the reference model and the scoreboard. If modifications must be made to either component, both may require restructuring. In addition, this reduces the degree of reusability, as the reference model and scoreboard behavior are tied together.

We propose that the reference model and scoreboard should be entirely decoupled, as shown in Figure 2. By separating the reference model and scoreboard, the object-oriented “Separation of Concerns” principle is respected, and both components become more focused and easier to maintain. If a new reference model is developed, the old reference model may be removed without replacing the scoreboard, reducing turnaround time.

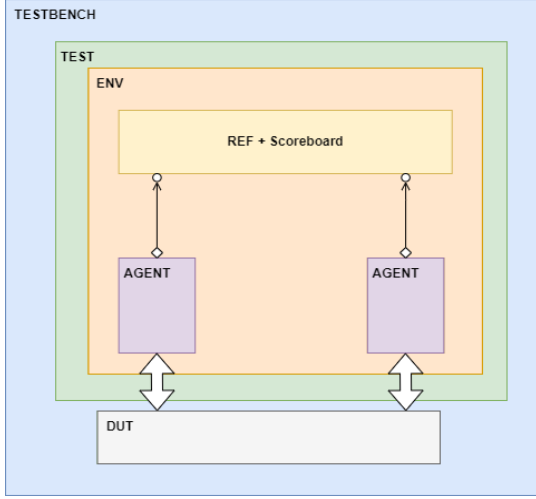


Figure 1. General UVM testbench setup where REF and Scoreboard are combined.

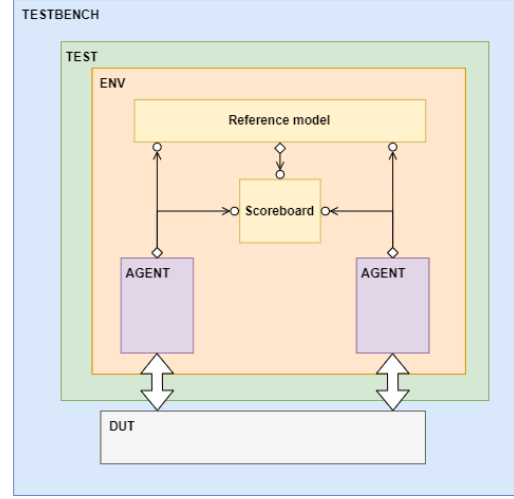


Figure 2. Proposed UVM testbench setup where REF and Scoreboard are decoupled.

The general structure when using the scoreboard is shown in Figure 3. A DUT is connected to two verification components (VC), both of which drive and receive transactions to/from the DUT. All transactions performed on the DUT are also passed on to the reference models, which in turn generate expected outputs. All inputs and outputs are fed into the scoreboard, where the DUT and each reference model have their own queues.

For each model used in the testbench, one queue is generated in the scoreboard. All items inserted in a queue are also tagged with some metadata indicating which producer they originated from. These producer tags are used when comparing items, as only items from the same producers ought to be compared. If e.g. the DUT is a NOC router, the producers may be one of the North, South, East, West and Local channel on the router.

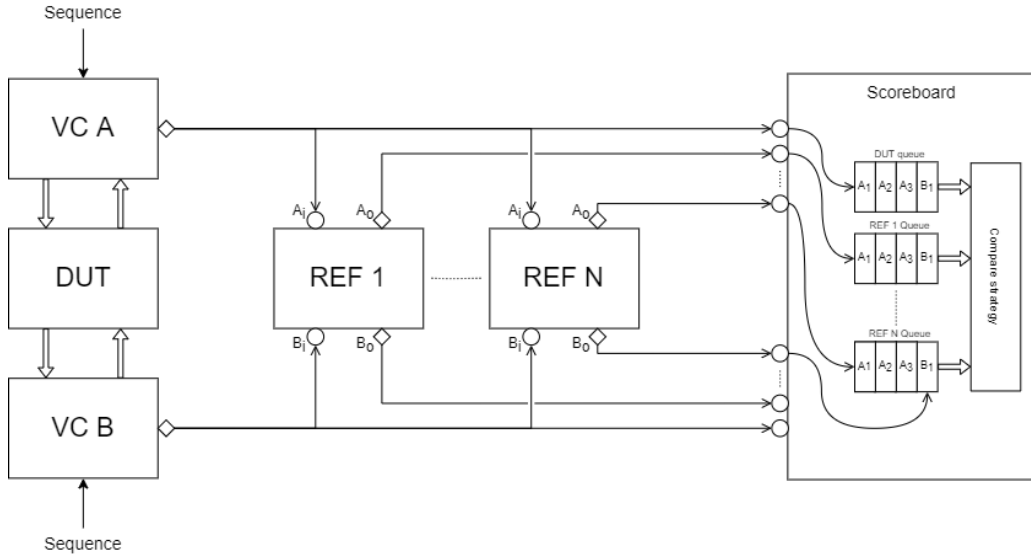


Figure 3. Structure of scoreboarding setup. A DUT has transactions driven onto it by two VC's which also receive transactions from the DUT. Transactions written to the two lowermost analysis ports on the scoreboard are also added to queues, but the arrows have been omitted for simplicity.

If the DUT does not model a bidirectional protocol where both requests and responses must be captured, but only a unidirectional protocol (such as a packet switch), the scoreboard setup shown in Figure 3 can obviously be simplified. In this case, VC A would only drive transactions and VC B would only receive outputs from the DUT.

Likewise, the reference models would only receive inputs from VC A, and the outputs would be compared to the output generated by VC B. An example configuration generating a similar testbench setup can be found in Figure 6.

This scalability of the scoreboard allows it to be used in many varied testbenches with different requirements. The scoreboard supports any number of DUT's and reference models and is easily configured with just a couple of configuration knobs, as shown in Section V.

### III. SCALABILITY AND ARCHITECTURAL SEPARATION

Our scoreboard can simultaneously interface to and compare any number of models: Design models (RTL, gate level), timed/untimed reference models (SystemVerilog, SystemC, Python), as well as physical devices like FPGA prototypes/ASICs. As a logical consequence, we assume a clear architectural separation between the models and the scoreboard implementation, the latter containing queues and comparison mechanisms, and we specifically choose not to employ the “expect” function concept.

#### A. Division of Tasks: REF vs SCB

A REF typically implements a transaction level model (TLM) of the RTL DUT, written in SystemVerilog, C/C++, SystemC or similar languages, and may be inherited from system modeling studies. A transaction level model does not model exact RTL pipeline and timing characteristics, and for some stimuli scenarios the order of transactions on the interfaces might differ between the models.

For each DUT pin level interface, the REF will have a transaction level interface. These interfaces might transfer information in and out of the device (e.g. a read/write SoC bus protocol), or only transport information in a single direction (e.g. a streaming based protocol). Depending on the exact goal of the complete verification environment, the reference model might model the full DUT functionality, or only parts of it. This depends on the ambitions for random stimuli, and what is practically possible to model in the transaction level reference model. For instance, a reference model of a packet switch might model the packet flow but defer from modeling the control flow (credits), as this would require the model to be fully or partially cycle accurate.

The SCB does not model the DUT. It merely queues the transactions going in and out of the DUT and the REF and is able to compare the activity of the models using a specific algorithm for comparing the data streams. This algorithm might range between a precise in-order and a relaxed out-of-order comparison, depending on how accurately the REF models the DUT's behavior (a cycle-accurate model might leverage In Order comparisons, whereas a functional model might use Out of Order comparisons). Methods implementing such comparison algorithms are a standard part of the SCB base class layers. Custom algorithms may be capable of analyzing individual transactions and put forward more specific requirements for transaction ordering. Such custom algorithms are easy to implement and employ in the SCB framework.

To summarize, the presented scoreboard architecture is capable of handling both uni- and bi-directional port traffic. Furthermore, the SCB is fully separated from one or more trailing reference models, allowing the use of REFs with same port topology as the DUT, and potentially modeled in different languages/domains than SystemVerilog. Also, the separation promotes reuse, e.g. by reusing module level SCBs at the SoC level.

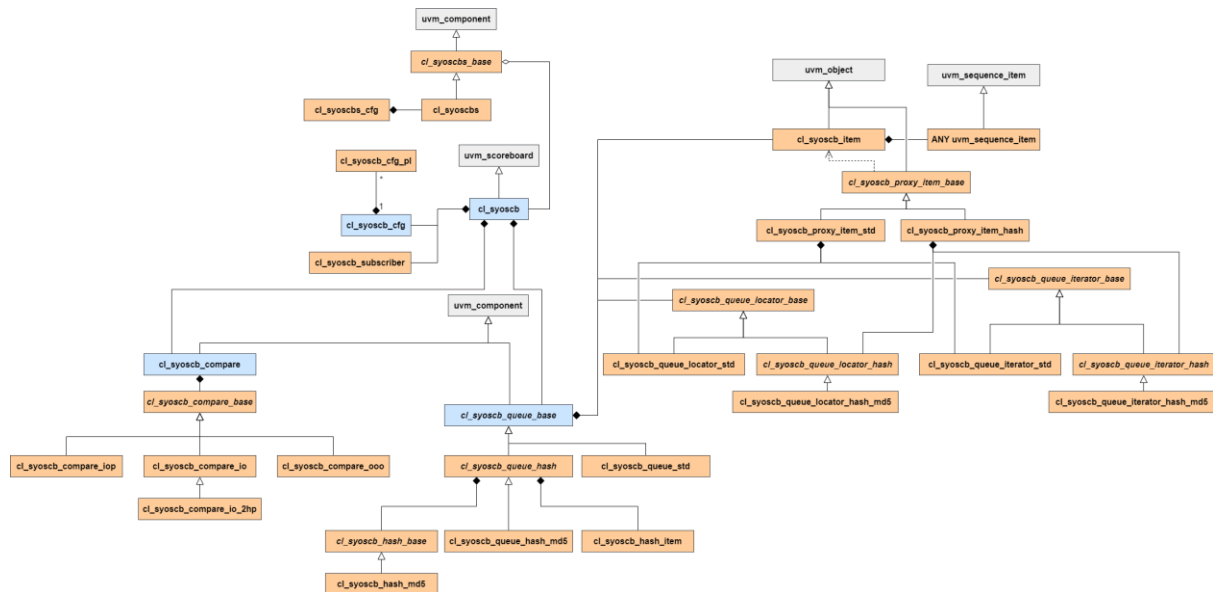
### IV. IMPLEMENTATION

The SCB structure, presented in Section II, includes a number of queues that matches the number of models used in the current testbench. Items in queues are tagged with some metadata, indicating which producer/port on a given model they were produced by. To use the scoreboard, items must be inserted either using a TLM-based connection, or via the function-based API, as explained below.

- TLM-based connections use TLM analysis ports to add items to the scoreboard. When creating the scoreboard, a UVM subscriber is generated for each producer on each queue. Writing items to these subscribers automatically wraps the item with metadata indicating which producer generated the item and inserts the item in the correct queue.
- The function-based API sidesteps the UVM subscribers and allows for direct insertion of items into the scoreboard. By passing a UVM sequence item, the queue and the producer, items can be added without use of a TLM connection.

Both mechanisms automatically wrap a given `uvm_sequence_item` with metadata and inserts it into the given queue. Note that while we propose one way of leveraging queues and producers (one queue per model, one producer per port on that model), it is up to the implementer to choose a meaningful use for the queue and item

producer concepts. It should be kept in mind that all pre-packaged compare methods delivered with the SCB architecture are designed to compare elements of the same producer type across all queues in the scoreboard.



Selected classes from the scoreboard are further explained below.

- **cl\_syoscb**: The scoreboard itself, extending the empty `uvm_scoreboard` base class. The scoreboard contains the queues and compare strategy used.
- **cl\_syoscb\_cfg**: The configuration object used for the scoreboard. Contains 30+ configuration knobs which can be used to fine-tune scoreboarding behavior. One configuration object should be generated for each scoreboard in the testbench – they cannot be reused across multiple scoreboards.
- **cl\_syoscb\_item**: This is the wrapper item inserted in the scoreboard, used to keep track of metadata for inserted sequence items. Each wrapper item has a handle to a `uvm_sequence_item`. This allows for easy integration into existing testbenches, as it does not require that all sequence items in a test inherit from `cl_syoscb_item`, but only that they inherit from `uvm_sequence_item`. The metadata includes the name of the producer, how many items have been inserted in the queue before that item, and how many items are currently ahead of it in the queue.
- **cl\_syoscb\_compare**: The base class for comparison algorithms. This class acts as the root of the comparison, using the OOP Strategy Pattern to delegate the actual comparison to a specific compare strategy. The scoreboard comes with four different comparison algorithms built in, which should be sufficient for most use cases. These are presented in Table 1. If another compare strategy is desired, this can be achieved by implementing the Compare Strategy API defined in `cl_syoscb_compare_base`.
- **cl\_syoscb\_queue\_base**: The base class for queues containing items. The scoreboard currently employs two different queue types. Standard queues (`cl_syoscb_queue_std`) use a generic SystemVerilog queue as the underlying data structure. These queues are good for in-order comparisons, where the items to be matched are at the front of the queue. Hash queues (`cl_syoscb_queue_hash`) use an associative array instead. Items are packed into a bitstream using the UVM *pack* function, and the bitstream is hashed to generate a checksum. Items are inserted into the associative array with their hash value as the key and the item as the value.

When using hash queues, items are not compared by comparing fields. Instead, the hash of one item is used to index into the associative arrays in all other queues. If an entry exists at that hash, it is assumed to be a match. Only in the exceedingly rare cases where a hash collision may occur are items compared field by field. Figure 5 presents a comparison of the time required to perform an out-of-order comparison of  $N$  items in a standard queue vs  $N$  items in a hash queue. As it is evident from the figure, hash queues are much more efficient than standard queues, especially for a large number of sequence items. Enabling the

configuration knob `ordered_next` ensures that queues are iterated over in insertion order. Enabling this feature incurs a slight performance hit but still performs much better than using standard queues. A default hash queue implementation using MD5 as the hash algorithm is supplied with the scoreboard. If other hashing algorithms are preferred, the API exposed in `cl_syoscb_hash_base` may be implemented.

| COMPARE ALGORITHM  | DESCRIPTION OF COMPARE ALGORITHM   |
|--|--|
| In Order<br>( <code>cl_syoscb_compare_io</code> )              | <p>Compares items in order, expecting an ordering in each secondary queue which is the exact same as in the primary queue.</p> <p>When a matching set is found, elements are removed from their respective queues. This will always be the first element of both primary and secondary queues.</p> <p>Error reporting: If the first element in a secondary queue does not match the first item in a primary queue.</p> <p>Note: Also provided is an in-order comparison algorithm optimized for 2 queues, named IO-2HP. The behavior is the same as the N-queue in-order comparison.</p> |
| In Order by Producer<br>( <code>cl_syoscb_compare_iop</code> ) | <p>Compares items in order, allowing out-of-order items so long as items from the same producer are in order. Thus, this is less strict than the normal in order compare.</p> <p>When a matching set is found, elements are removed from the respective queues. These may not be the first items in their queues but will be the first item from a given producer.</p> <p>Error reporting: If the first element in a secondary queue of a specific producer type is different from the first element in the primary queue of the same producer type.</p>                                 |
| Out of Order<br>( <code>cl_syoscb_compare_ooo</code> )         | <p>Compares elements out of order, making no assumptions about the specific ordering of queues. Used to check that each secondary queue contains the same contents as the primary queue, but without any specific ordering restrictions.</p> <p>When a matching set is found, elements are removed from their respective queues.</p> <p>Error reporting: If a queue grows larger than a set threshold. Does not generate an error if a matching item cannot be found.</p>  |

Table 1: Prepackaged compare methods.

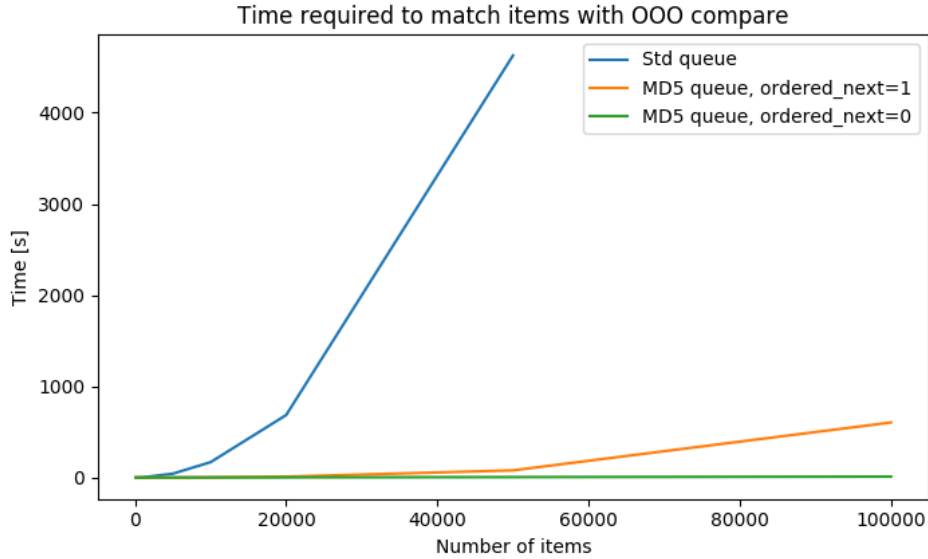


Figure 5. Comparison of runtime when performing OOO compare with different queue topologies. Using hash queues is much faster than using standard queues. Benchmarks were run on Cadence Xcelium version 21.09.003.

## V. EXAMPLE CONFIGURATION

To aid in simplifying scoreboard setup, each SCB is associated with a configuration object. This configuration object contains several knobs that can be used to control the scoreboard, both prior to and while running a test. Initializing a configuration object, specifying the scoreboard name and attached queues and producers is as simple as executing the `cl_syoscb_cfg::init` method with the correct parameters, and then passing the configuration object to the scoreboard via the UVM configuration database.

In the UVM build phase of a test or environment, the scoreboard configuration object should be created. This is shown in Listing 1. Once forwarded to the environment, the environment may instantiate the scoreboard, forwarding the configuration object to the scoreboard. In the environment's UVM connect phase, the generated subscribers may then be used to create a TLM connection to the DUT and reference models, as shown in Listing 2. The subscribers will then automatically insert received items into the scoreboard.

```
function void my_test::build_phase(uvm_phase phase);
    cl_syoscb_cfg cfg = cl_syoscb_cfg::create("cfg"); //Create config
    cfg.set_compare_type(pk_syoscb::SYOSCB_COMPARE_IO); //Use in-order comparisons
    cfg.set_queue_type(pk_syoscb::SYOSCB_QUEUE_STD); //Use standard SV-queues
    cfg.init("SCB 1", {"REF", "DUT"}, {"P1", "P2"}); //Initialize 1 SCB named "SCB 1", two queues
                                                    //(REF and DUT), and producers P1, P2 for both
    uvm_config_db #(cl_syoscb_cfg)::set(this, "scb_env", "cfg", cfg); //Forward config to environment
    this.scb_env = my_env::type_id::create("scb_env"); //Create env which will get cfg and create scb
endfunction: build_phase
```

Listing 1. Creating a scoreboard configuration object and forwarding it to the environment.

```
function void my_env::connect_phase(uvm_phase phase);
    uvm_subscriber dut_p1_sub, dut_p2_sub;
    //Get subscribers for DUT, producer P1 and P2
    dut_p1_sub = this.scb.get_subscriber("DUT", "P1");
    dut_p2_sub = this.scb.get_subscriber("DUT", "P2");
    //Connect to DUT monitor's P1 and P2 analysis exports
    this.dut_mon.p1_port.connect(dut_p1_sub.analysis_export);
    this.dut_mon.p2_port.connect(dut_p2_sub.analysis_export);
    //do the same for remaining subscribers and producers
endfunction: connect_phase
```

Listing 2. Connecting a monitor to scoreboard subscriber.

The above configuration would lead to a scoreboard setup like the one shown in Figure 6. This shows how easily the scoreboard may be set up. Running the simulation will now automatically insert items into the scoreboard, and the scoreboard will compare items whenever both queues contain sequence items.

Table 2 presents some additional configurations knobs present in `cl_syoscb_cfg`. Note that the table is not exhaustive, and many more configuration knobs are present, as further described in the documentation included with the scoreboard.

| CONFIG KNOB   | PURPOSE  |
|---|--|
| void set_max_queue_size(<br>string queue_name,<br>int unsigned mqs)                               | Per-queue limit for the maximum number of elements that can be in that queue. Issue a UVM ERROR if exceeded.   |
| void set_max_search_window(<br>string queue_name,<br>int unsigned msw)                            | Per-queue value determining the max number of elements that should be tried in each queue when performing OOO compare with standard queues. If 0, all elements in the queue are tried. (Hash queues bypass this by using the hash value to index into an associative array). |
| void set_comparer(<br>uvm_comparer comparer,<br>string queue_names[],<br>string producer_names[]) | For all combinations of given queue names and producer names, use that comparer when comparing items. Allows for fine-tuned comparison behaviour.  |
| void set_scb_stat_interval(<br>int unsigned ssi)  | Set an interval such that insertions, matches, flushes and remaining items in the SCB are printed after every N insertions into the scoreboard.  |
| void set_disable_compare_after_error(<br>bit dcae)  | Toggle whether the SCB should be able to proceed without comparing items once a UVM ERROR has been detected  |

Table 2. Examples of setter methods for configuration knobs. Not an exhaustive list of configuration knobs in the scoreboard.

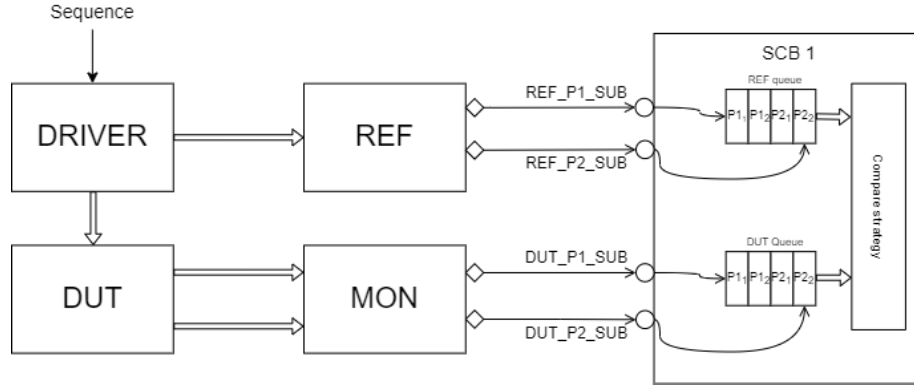


Figure 6. Scoreboard setup generated with the configuration code in in Listings 1 and 2.

## VI. DEBUGGING FEATURES

The goal of this scoreboard is, in addition to removing the need for re-inventing the scoreboard for every new verification project, to provide helpful debugging features which can ease the debugging process when errors invariably appear in a simulation. This section highlights a number of the debugging features that the scoreboard provides.

### A. Common error checks

To reduce the chance of human errors impacting a test, the scoreboard comes with some features that serve to remove common errors. All these checks may obviously be enabled/disabled using a configuration knob if so desired.

- **No Items Inserted Check** issues an error if a scoreboard had no insertions during a test. Forgetting to connect all components in a UVM testbench is not unheard of, and this simple check ensures that an error is generated the first time the test is run, allowing the engineer to quickly remedy the error.
- **Item isolation** is used to ensure that sequence items are not modified after insertion into the scoreboard. By automatically cloning all sequence items inserted into the scoreboard, it becomes impossible to alter the items after insertion. This ensures that potential matches do not become invalidated due to human error. Note that this requires that the UVM `copy / do_copy` methods have been properly implemented, as the item isolation features leverages these methods.
- **Orphan detection** verifies that no items remain in the queues once simulation finishes. If items do remain, an error is generated, and orphan information is printed to the terminal and may also be dumped to a logfile as described below.

### B. Transaction streaming

The scoreboard supports streaming transactions out of the scoreboard for runtime or post-simulation analysis. Two types of transaction streams are supported.

The first, a full scoreboard dump, dumps every transaction coming into the scoreboard to a logfile. It may either dump all transactions into the same file, or it may dump them to separate files, one for each queue. Configuration knobs in `cl_syoscb_cfg` determine the rate at which items are logged to a file – ranging from logging every item as soon as it is inserted, to keeping all sequence items in memory until the end of simulation. By streaming items to a file during simulation, a run-time analysis script may be used to monitor the current test. After simulation, the files may be analyzed by a post-simulation script.

The second type of transaction stream is the “orphan” dump. When simulation finishes, items may remain in some of the queues. These items are dubbed orphans, and may be logged to separate files, one for each queue. Orphan dumping is controlled by the configuration knob `dump_orphans_to_files`.

In both cases, sequence items may be logged to a TXT file, or they may be dumped to an XML file, using the provided XML printer (`uvm_xml_printer`). Dumping to a TXT file provides a quick reference that can be skimmed to look for certain transactions, whereas XML files allow for automated analysis or transformation into

other formats. The scoreboard comes with two XSLT files for transforming the dumped XML files into either GraphML or HTML for visual inspection. With the Makefile included in the release, transforming an XML file to HTML is as simple as executing the command.

```
make generate_html XML_FILE={FILENAME.XML} HTML_OUT_FILE={FILENAME.HTML}
```

### C. Miscompare tables

If two items in the scoreboard fail a comparison check, this is an indication that the DUT and REF do not agree. To easily identify which fields of the sequence items that did not match, the scoreboard employs a miscompare table to show the two items side by side, as shown in Figure 7. The table shows the items involved in the comparison and includes information from the `uvm_comparer` used for the comparison on which fields did not match.

```
#####
# [syoscb0]: cmp-io: Item from primary queue (Q1) not found in secondary queue (Q2) #
#####
```

| Name            | Type           | Size | Value      | Name            | Type           | Size | Value      |
|-----------------|----------------|------|------------|-----------------|----------------|------|------------|
| P1-item-74      | cl_syoscb_item | -    | @74        | P1-item-4284    | cl_syoscb_item | -    | @4284      |
| insertion_index | integral       | 64   | 'd2        | insertion_index | integral       | 64   | 'd2        |
| queue_index     | integral       | 64   | 'd0        | queue_index     | integral       | 64   | 'd0        |
| producer        | string         | 2    | P1         | producer        | string         | 2    | P1         |
| item            | large_seq_item | -    | @4293      | item            | large_seq_item | -    | @4344      |
| int_a           | integral       | 32   | 'ha4efadcf | int_a           | integral       | 32   | 'h73d1ae30 |
| int_b           | integral       | 32   | 'h0        | int_b           | integral       | 32   | 'h0        |
| int_arr         | da(integral)   | 10   | -          | int_arr         | da(integral)   | 7    | -          |
| [0]             | integral       | 32   | 'hfb36eb   | [0]             | integral       | 32   | 'h654dc585 |
| [1]             | integral       | 32   | 'h2829615b | [1]             | integral       | 32   | 'hda82ea35 |
| [2]             | integral       | 32   | 'h2c87f169 | [2]             | integral       | 32   | 'hc8384aee |
| [3]             | integral       | 32   | 'h5efe1bef | [3]             | integral       | 32   | 'h6864a3a2 |
| [4]             | integral       | 32   | 'h7e9254b4 | [4]             | integral       | 32   | 'he7fb5a61 |
| [5]             | integral       | 32   | 'he863cea4 | [5]             | integral       | 32   | 'h9aa9601c |
| [6]             | integral       | 32   | 'hc14ee214 | [6]             | integral       | 32   | 'hfab15e5e |
| [7]             | integral       | 32   | 'hce0f42ec |                 |                |      |            |
| [8]             | integral       | 32   | 'h1e38c0ab |                 |                |      |            |
| [9]             | integral       | 32   | 'h1448609c |                 |                |      |            |

```
#####
Results from uvm_comparer::get_miscompares() [show_max=5]
-----
P1-item-4284.item.int_a: lhs = 'h73d1ae30 : rhs = 'ha4efadcf
P1-item-4284.item.int_arr.size: lhs = 'h7 : rhs = 'ha
#####
```

Figure 7. Miscompare table. The table shows the sequence items that failed comparison, as well as which fields of the sequence items did not match.

This feature can significantly improve debugging capabilities for failed tests, and we find that it provides a much more intuitive and efficient way of resolving failed comparisons, as all necessary information is presented in a structured manner. We are not aware of any other UVM scoreboards that provide this feature.

## VII. IMPROVEMENTS SINCE LAST PUBLICATION

The scoreboard has seen a large number of improvements since the last publication. This section highlights a number of the most important improvements.

### A. Multiple scoreboard instances

To aid in scenarios where multiple similar scoreboards may be necessary, we also provide a scoreboard wrapper, `cl_syoscb`s which can be used to generate multiple identical or near-identical scoreboards. This wrapper has been tested in a project with 1000+ wrapped scoreboards, where it proved to increase productivity as the scoreboard wrapper simplified setting up and interfacing with all wrapped scoreboards.



### B. Miscompare tables

Miscompare tables, presented in Section VI.C, are one of the major improvements since the last release, as they dramatically simplify the debugging process. The feature has already been used in internal projects where it has helped to more quickly identify and resolve problems.

### C. Additional metadata

In the first release of the scoreboard, the only metadata tracked in `cl_syoscb_item` was the producer that had generated a sequence item. In the new version, this has been expanded to also track an item's insertion index and queue index. An insertion index of  $N$  indicates that an item is the  $N$ 'th item inserted in a given queue. The queue index is only used when dumping orphans and indicates an item's relative position in the queue. Queue indexes are not updated throughout simulation, as it is not feasible to update potentially thousands or millions of indexes when an item is removed from a queue.

### D. Support for UVM IEEE

The latest release of the scoreboard works with UVM 1.1d, UVM 1.2 and UVM IEEE-versions without any additional configuration. This makes it very simple to use the scoreboard in multiple different simulation setups.

### E. Other improvements

By profiling the performance of the scoreboard, bottlenecks have been identified and removed from the scoreboard. In some instances, such as when iterating over very large queues, the speedup is on the order of 1.25 – 2x, drastically reducing the time required for simulation.

The documentation for the scoreboard has been updated, and now includes more details and in-depth explanations of how the scoreboard works. More unit tests have been added, and these are also described in the documentation such that they may serve as a starting point for anyone wanting to implement the scoreboard in their own verification environment.

The latest version of the scoreboard has been tested on the following simulators:

- Synopsys VCS® version XX **TODO fill in version number**
- Siemens EDA Questasim® version 21.4
- Cadence Xcelium® version 21.09.003
- Cadence Incisive® version XX **TODO fill in version number**

## VIII. CONCLUSION

In this work we propose an industry proven, scalable UVM scoreboard architecture, able to interface to any number of design models across languages, methodologies, abstractions and physical form. Any relationship between data streams can be checked using pre-packaged and custom compare methods, and we make it easy to interface external checker and debug aiding applications. Based on our work, the SV/UVM user ecosystem will be able to improve how scoreboards are designed, configured and reused across projects, applications and models/architectural levels.

## IX. AVAILABILITY

The UVM scoreboard has been released for general availability, and can be downloaded from the following locations:

- Accelera UVM Forum
  - **Insert Link**
- SyoSil Website:
  - **Insert Line**

The release features the UVM Scoreboard, 70+ testcases, thoroughly documented tests and code, as well as examples, release notes and documentation.

The scoreboard has been released under the Apache 2.0 license and can be used freely. Any suggestions for how to improve the base classes and examples are very welcome, including potential bug reports. Please direct such feedback per email to the authors at [scoreboard@syosil.com](mailto:scoreboard@syosil.com)

## X. REFERENCES

- [1] Accellera, "UVM Resources," [Online]. Available: <https://forums.accellera.org/files/category/3-uvm>. [Accessed 6 4 2022].
- [2] J. Andersen, P. Jensen and K. Steffensen, "Versatile UVM Scoreboarding," in *DVCon*, 2015.
- [3] Mentor Graphics, "Verification Academy," [Online]. Available: <https://verificationacademy.com/topics/verification-methodology/uvm-connect>. [Accessed 16 03 2022].
- [4] Accellera, "SCE-MI (Standard Co-Emulation Modeling Interface)," Accellera, 06 12 2016. [Online]. Available: <https://accellera.org/downloads/standards/sce-mi>. [Accessed 16 03 2022].