Versatile UVM Scoreboarding

Jacob Andersen, Peter Jensen, Kevin Steffensen

SyoSil ApS, Copenhagen, Denmark

{jacob, peter, kevin}@syosil.com

Abstract — **All UVM engineers employ scoreboarding for checking DUT/REF behavior, but only few spend their time wisely by employing an existing scoreboard architecture. Main reason is that existing frameworks have inadequately served the user needs, and have failed to accelerate the user efficiency in the debug situation. This paper presents a better UVM scoreboard framework, focusing on scalability, architectural separation and connectivity to foreign environments. Our scoreboard architecture has successfully been used in UVM testbenches at various architectural levels, across models (RTL, SC) and on physical devices (FPGA/ASICs). 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 contain lots of common denominators of deficiency. Users struggle to implement checker mechanisms for the designs under test being exposed to random stimuli, while sacrificing aspects like efficiency, easy debug and a clean implementation.

Existing user donated work [1] present 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 is able to check the design under test (DUT) 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 existing proposals to inadequately 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).

II. Scalability & architectural separation

Our scoreboard is able to simultaneously interface 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.

To simplify text and schematics, the sections below explain the architectural separation between a DUT and a reference model (REF), tailored to check the DUT running random stimuli. In subsequent sections, we will

present how the scoreboard (SCB) interfaces to other models and physical devices, and compares the operation of any number of models.

A. *Division of Tasks; REF vs SCB*

A REF typically implements a transaction level model 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 packet 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 running random simulations and what practically is 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 compare, depending on how well the reference model is DUT accurate. 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.

B. *Implementation Details*

Figures 1 and 2 figure present the structure and interconnect of the REF and the SCB. We here have multiple REFs to show how the solution scales. The DUT is the primary model. This is determined by the DUT being attached to the verification environment that creates the design stimuli with UVM sequences. The verification components (VCs) drive the DUT stimuli, and from the VC analysis ports, transactions (UVM sequence items) are picked up by one or more REFs (M1 … Mn) as well as the SCB. The REFs are trailing secondary models, as these are unable to proceed with execution before the primary model stimuli have been created, applied, and broadcast by the VC monitor. The REFs connect to the VCs using analysis FIFOs, allowing the REF to defer handling the transaction until potential dependencies have been resolved on other ports.

When considering a SoC bus interface with the VC as initiator, the transactions received by the REFs contain both a *req* part (request, DUT stimuli) and a *rsp* part (DUT response), as we do not expect the UVM environment to offer an analysis port sending only the *req* part. The REF needs the *req* part, and will replace the *rsp* part with a computed response, based on the *req* and the REF internal state. For instance, the *req* part can be a bus read request (address and byte enables), whereas the *rsp* part then will be the data read from the DUT. It is mandatory for the REF to create its own transaction (e.g. by using the sequence item copy method), as altering the transaction received from the analysis port will also alter the transaction received by the SCB from the DUT.

For each model (M1 … Mn) attached to the scoreboard, any number of SCB queues can be handled. Each queue contains meta-transactions, wrapping UVM sequence items along with metadata. This allows same queue to contain different transaction types not necessarily comparable with each other. The metadata ensure that only queue elements of same type are compared.

The queues can be organized in several ways. The SCB in the figure displays the normal configuration; one queue per model, with different transaction types in the queues. The generic compare methods only compare transactions of same type (A, B) between the queues. Organizing one single queue per model is simple to configure, and provides great help when debugging failing scenarios, as a timeline with the full transaction flow can be visualized. The alternative queue configuration shown in the figure is one queue for each port for each model. For some application types it might be desirable to configure the queues in this fashion, if the relationship between the ports is irrelevant for debugging the SCB queues.
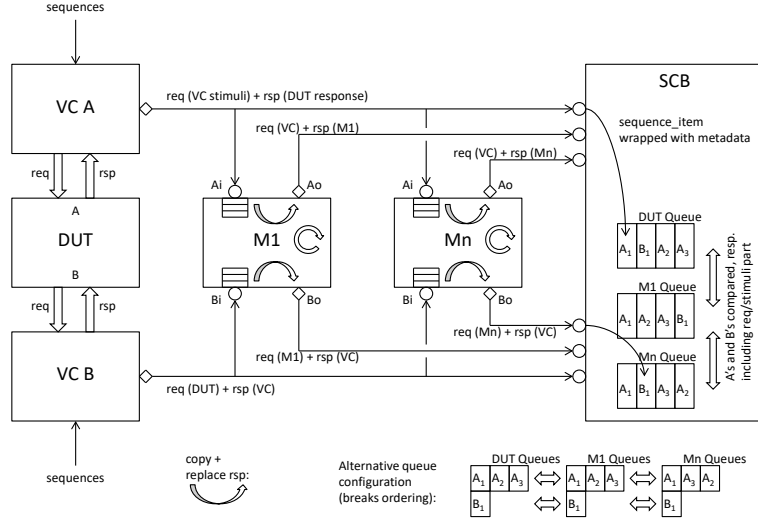
Figure 1. DUT, trailing models and scoreboard architecture.
Bi-directional information flow on DUT ports. Configuration suitable for SoC bus interfaces.

The generalized REF/SCB configuration in Figure 1 can be simplified to Figure 2 for the special case where the DUT ports only employ a uni-directional information flow, e.g. a packet based traffic pattern. No DUT/Mx response on the A ports and no testbench stimuli on the B ports occurs. Hence the structure resembles more traditional scoreboard usage, where a DUT packet stream simply is compared to the REF packet stream. In the figure we show how the stimuli (A port) still can be added to the SCB queues. This will not add value to the performed checks, but will greatly aid the debug situation, as the queues will present both input and output DUT/REF traffic.
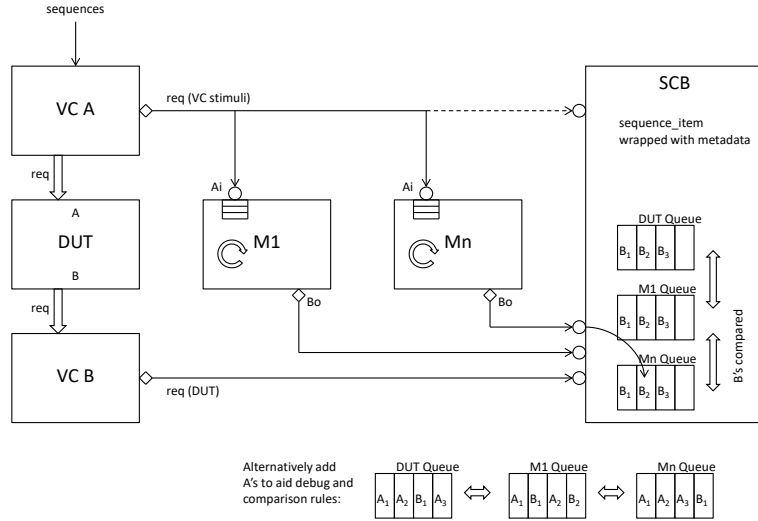

Figure 2. DUT, trailing models and scoreboard architecture.
Uni-directional information flow on DUT ports. Configuration suitable for packet based flow.

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.

III. NON-UVM CONNECTIVITY

Besides interfacing to UVM using analysis ports, establishing links to non-UVM/non-SystemVerilog code is essential to keep the scoreboard versatile and reusable, enabling the use of external checkers and debug aiding scripts. For this purpose, the scoreboard framework offers a number of external interfaces:

| Interface	Language	Execution	Purpose/Benefits
VP	SystemC	Run-time	Reference model (virtual prototype)
Queue	SystemC/C++	Run-time	Use for high performance queue comparison, low memory footprint Use for interface to existing C/C++ protocol checkers
App Sockets	Python	Run-time	Use for creating checkers and debug aiding scripts, easily creating complex data structures and debug aiding scripts.
	Custom	Run-time	Use for interfacing to any language callable from C++
Logger	XML / TXT	Post-sim	Streaming socket to log file, XML and/or TXT Usable for post-simulation analysis purposes
Non-UVM	SV	Run-Time	Interface to non-UVM SystemVerilog interfaces (not depicted in below figure).

Table 1: Non-UVM Interfaces
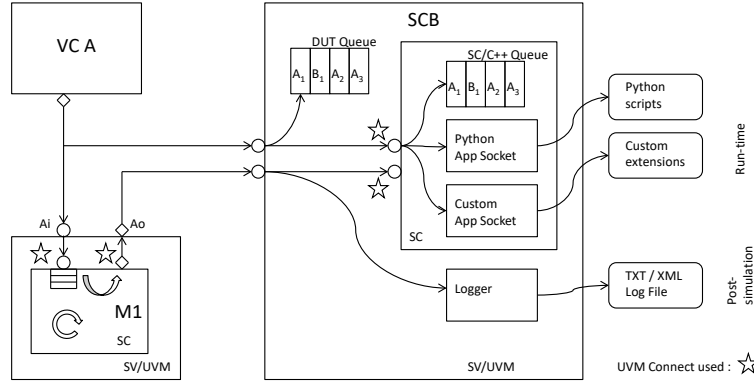
Figure 3. Interfacing to SystemC REF using UVM Connect.
Interfacing to SystemC/C++ SCB Queues, Python and Custom App Sockets, XML Stream Socket.

The UVM Connect library enables seamless TLM1/TLM2 communication between SystemVerilog/UVM and SystemC [2]. We employ the library for implementing most run-time interfaces listed above. For connecting analysis ports between SV/SC, we employ uvmc_tlm1 sockets with generic payloads, using the "sv2sc2sv" pattern where SV and SC both act as producer/consumer. To use this pattern, pack/unpack methods must be available on both sides of the language boundary. Today we use the UVM field macro generated pack/unpack methods in SV, and the UVMC macros for the SC side. If performance issues arise in a specific implementation, a shift to using the dedicated UVM pack/unpack macros is implemented.

A. Streaming Transactions out of SCB

To allow external resources to receive a transaction stream, we offer interfaces both in SC/C++ and Python. These interfaces are mainly intended to be used run-time, such that external scripts are being evaluated while the SV simulation is running, avoiding creating large log files for post-simulation processing.

For the Python App Socket, the user has to manually implement the pack/unpack methods on the Python side. Once done, the user can write Python scripts with analysis ports, where a function is called every time a sequence item is received by the SCB. Hence the user can attach extern Python scripts with analysis capabilities not present in SystemVerilog – either if too difficult to model – or if existing Python analysis scripts are available. Furthermore Python is an easy bridge towards other tools, where run-time streaming of SCB activity is needed.

B. Streaming Transactions into SCB

The scoreboard can be used with transactions streams from external resources, e.g. by obtaining logs from devices running in the lab (silicon, FPGA, emulators). Depending on the log format, we use either the XML

Interface or the Python App Socket to retrieve the log transactions as UVM sequence items. In this configuration, the SCB will implement an analysis port, for sending the transactions to REF(s) and then compare the device simulation log with the REF (figure below). In this configuration, the external model acts as the primary model, the M1 REF model is a trailing secondary model. Obviously, the UVM environment is passive and does not run any sequences to create stimuli. The preferred configuration is to stream the external device log through the SCB external interfaces while the device is running, to avoid post-simulation analysis of large log files.
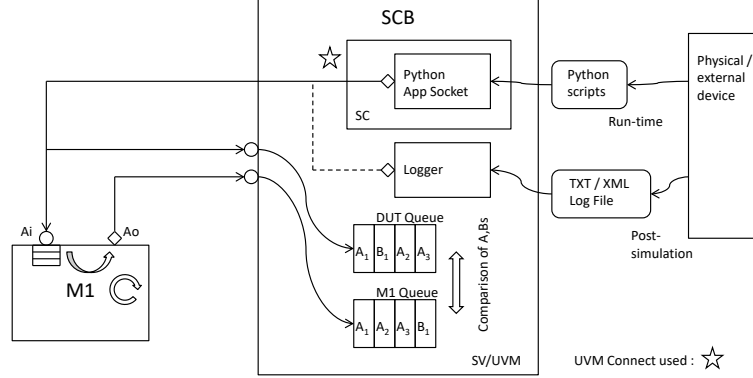

Figure 4. Scoreboarding XML transaction stream from external device, compare against REF.

We have considered connecting the SCB to external physical targets using the SCE-MI standard [3], mainly being relevant if the drivers/monitors in the UVM testbench are synthesizable. In this situation, a fast-running connection to the UVM scoreboard is rather easy to set up, and interfacing using XML/Python would be less required. Exploring this solution is unfortunately out of the scope of this paper.

IV. IMPLEMENTATION & CONFIGURABILITY

The SCB implements a generic scoreboard architecture, which is organized in any number of queues. In turn, each queue is able to contain items from different producers, indicated by tagging with different meta-data. This forms a two-dimensional structure with MxN entries (refer to Figures 1 and 2).

For configuring the SCB, a configuration class offers with a dedicated API, making it easy to configure the number of queues and producers. For populating the SCB with traffic from the DUT and REFs, two different APIs can be used to insert item into the queues:

- Transaction based API: Insertion by using the automatically generated UVM analysis exports. Used for the normal UVM use cases.

- Function based API: Insertion by calling the add_item method. Used if transactions require transformation before insertion, e.g. if not extended from uvm_sequence_item, or if hooking up as callback. See section V for example.

Both mechanisms automatically wrap a given uvm_sequence_item with meta data and inserts it into the given queue. Thus, there can be any relationship between the number of ingoing transaction streams and the number of queues and item producers in the scoreboard. It is up to the implementer to choose a meaningful use for the queue and item producer concepts, while keeping in mind that all pre-packaged compare methods delivered with the SCB architecture aims at comparing all elements of same item producer type across all queues.

The SCB continuously evaluates the contents of the queues, using the configured method to remove matching entries that satisfies the criteria of the compare method, as well as reporting whenever elements in the queues violate the compare rules. Models produce transaction flows in different orders. The scoreboard allows specification of the evaluation criteria, triggering when a comparison between a set of queues is evaluated. This is done by promoting a specific queue to become a primary queue, leaving the secondary queues to be trailing. A time consuming RTL model will typically be set as primary mode, whereas a zero-time transactional reference

model will be set as secondary queue. Furthermore, evaluation will also be triggered at end-of-simulation to ensure that all queue contents are as expected.

The generic scoreboard architecture is implemented by extending standard UVM base classes. This allows us to use the UVM Factory to specialize a scoreboard implementation, e.g. by changing the comparison algorithm for a specific test. In the below figure, the UML diagram of the scoreboard UVM classes is shown. Only selected classes will be described in the sections below.

Figure 5. UML diagram: Scoreboard Classes.

- **cl_syoscb:** The scoreboard itself is an extension of cl_syoscb, in turn an extension of the basic empty uvm_scoreboard.

- **cl_syoscb_queue:** The class only provides an API for implementing a queue (virtual class) and it does not provide the actual implementation. This abstraction allows changing the implementation of how the queues are searched. One implementation of cl_syoscb_queue is offered in the cl_syoscb_queue_std class, where the queues are not implemented directly as SystemVerilog queues of cl_syoscb_item objects. Rather, we employ SystemVerilog associative arrays (hashes) for modeling the queues. The hash key is the checksum computed from the string representation of the item. The use of associative arrays speeds up searching for specific transactions in the queues, and allows accelerated comparison of transaction pairs using the checksums. Ordering information is kept with the metadata as the comparison might need this information. In this way, the scoreboard implementation offers a SystemVerilog queue like view of the SCB queues, such that the user avoids working with associative arrays and checksum keys. Rather, the user can use the offered an iterator/locator mechanism to search, traverse and compare the queues, with good SCB performance.

- **cl_syoscb_item:** The SCB items are not derived directly from uvm_sequence_item since this would enforce that all transaction classes in a given testbench would have to be inherited from cl_syoscb_item. This would make it difficult to use the SCB in an already existing testbench. To overcome this problem and provide easy adoption and integration in e.g. existing testbenches, the OOP Decorator pattern is used for linking the meta-transaction (cl_syoscb_item) to the underlying uvm_sequence_item. Before sequence items from a VC monitor analysis port are inserted into the queues, the sequence items are wrapped in the meta-transaction cl_syoscb_item class together with meta data specific for each transaction, e.g. the checksum key and the name of the producer.

- **cl_syoscb_compare:** The compare method is encapsulated by the cl_syoscb_compare class, utilizing the Strategy OOP design pattern. This class instantiates via a factory lookup an instance of the cl_syoscb_compare_base class. The cl_syoscb_base class is an abstract class with only virtual methods, it provides and enforces an API for compare methods with makes it fairly easy to implement custom compare methods. The SCB comes with several "ready to use" compare methods as listed in Table 3. The default compare is the in order by producer (cl_syoscb_compare_io_producer).

Compare Method	Description of Compare Algorithm
Out of Order (cl_syoscb_compare_ooo)	Performs a 1:1 element out of order compare across all queues. Used to check that each secondary queue contains same contents as the primary queue, but without any specific ordering restrictions. When a matching set is found, elements are removed from the respective queues. Error reporting: If a queue grows above a set threshold, or a queue timeout happens.
In Order (cl_syoscb_compare_io)	Performs a 1:1 element in order compare across all queues. Used to check that each secondary queue contains same contents as the primary queue, and that the ordering is exactly the same, also regarding the producer types. When a matching set is found, elements are removed from the respective queues. This will always be the first element of both primary and secondary queues. Error reporting: If the first element in a secondary queue is different from the first element in the primary queue, disregarding the producer type. Also if a queue threshold/timeout happens.
In Order by Producer (cl_syoscb_compare_io_producer)	Performs a 1:1 element in order compare across all queues. Used to check that each secondary queue contains the same contents in the same order as the primary queue but only within the same producer. Thus, this is less strict than the normal in order compare. When a matching set is found, elements are removed from the respective queues. This will always be the first element of the primary queue. 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. Also if a queue threshold/timeout happens.

Table 2: List of prepackaged compare methods

- **cl_syoscb_cfg:** The SCB utilizes the UVM configuration database such that it can be reconfigured on the test case level. This allows changing e.g. the number of queues and compare algorithms. For instance, a user extension of cl_syoscb_cfg can be used for this purpose:

```
class cl_scb_myconfig extends cl_syoscb_cfg;
  function new(string name = "cl_scb_myconfig");
    this.set_queues({"RTL", "M1"});
    this.set_primary_queue("RTL");
    this.set_producer("A", {"RTL", "M1"});
    this.set_producer("B", {"RTL", "M1"});
  endfunction
endclass
```

A configuration class has to be provided as a default one will not be created. The example above shows how two queues named RTL and M1 are configured (with two producers: A and B). Table 3 below shows the full list of configuration options. The options can be set or read by using a standard get/set API. The compare method is separately configured by using the UVM factory.

Attribute	Type	Description
queues	string -> cl_syoscb_queue	Stores the relationship from queue name to actual queue
producers	string -> string[]	Store the relationship from producer name to the list of queues where the producer is present
primary_queue	string	Configures the primary queue name. Compare methods use the primary queue as trigger for when to execute its algorithm.
full_scb_dump	bit	Turns full scoreboard dump on or off. This is also controllable from the command line by adding "+uvm_scb_fsd=1" to the command line.
max_queue_size	string -> int unsigned	Controls the maximum size of a given individual queue.
max_full_queue_size	string -> int unsigned	Specifies the threshold of when a given queue is dumped to file.
full_scb_type	string[]	Specifies the type(s) of the full SCB dump. TXT/XML is supported.
item_timeout_queue	string->int unsigned	Set a per queue timeout on items.
item_timeout_producer	string->int unsigned	Set a per producer timeout on items.

Table 3: List of scoreboard configuration attributes

Once the SCB is properly configured a standard uvm_sequence_item easily can be inserted into the SCB without manually managing the meta data. In the below example, the verification environment uses the transaction based API to retrieve the analysis export for connection with the analysis port of the verification component:

```
cl_scb_uvm scb;
…
myvc.ap.connect(scb.get_aexport("RTL", "A"));
```

Compare method configuration is done easily by a factory overwrite, e.g. on the test level:

```
cl_syoscb_compare_base::set_type_override_by_type(
                         cl_syoscb_compare_base::get_type(),
                         cl_syoscb_compare_ooo::get_type(),
                         "*");
```

If the already provided compare methods do not fit the needed compare scheme, then a custom compare can be implemented by extending the cl_syoscb_compare_base class. For easing the implementation, each queue has a locator object attached which provides a collection of search algorithms for traversing the queues in an elegant and easy manner. Additionally, standard iterator objects are also available for iterating through either search results or selected parts of a queue.

V. Flexibility

Access to model state information may be required to compare transaction flows from different models. The scoreboard generalizes this, using designated abstract channels, in turn enabling easy exchange of models. This is done by only adapting glue code, in order to attach the designated channel to a new model.

Furthermore, the SCB can also be used in non-UVM environments, e.g. VMM, by utilizing the function based API. For instance, a VMM transactor can be implemented which creates a uvm_sequence_item and inserts it into the SCB. For this purpose the SCB provides the uvm_sequence_item_vmm which wraps a vmm_data class:

```
class xactor extends vmm_xactor;

    vmm_channel chan;

    cl_syoscb scb;

    vmm_data data;

    uvm_sequence_item_vmm item;

…

    chan.get(data);

    item = new();

    item.data = data;

    scb.add_item("RTL", "A", item);  // queue, producer, sequence item
```

VI. DEBUG

Most scoreboards just echo the difference between the expected and the actual transaction flow. Our scoreboard architecture offers mechanisms for understanding the full transaction flow and model context at the point of failure. The scoreboard offers two ways of debugging aid:

Logging: During normal simulations, the scoreboard keeps down the queue sizes by evaluating the instantiated compare method. When an error happens, the remaining queue content is written to the simulation logs, displaying the "diff" of the queues at the point of failure. With this information, it is difficult to diagnose the cause of the error post-simulation, as the output does not contain the full simulation history. By enabling the "full scoreboard dump" feature (Table 2) all elements added to the scoreboard queues during the simulation will be dumped to a set of files. The dump mechanism dumps to text format or XML format depending on the configuration of the scoreboard. Also, XLST transformations can convert the transaction dumped XML into other file formats, e.g. GRAPHML XML to produce timed transaction graphs.

Analysis: Using the APIs described in section III, external verification scripts can get run-time access to the transaction streams, aiding the debug process by analyzing the streams and producing higher-order views of the traffic. A good example would be to visualize the enumeration process on a PCIe bus system. Analysis scripts can be implemented to read the dumped XML file or access the transactions at runtime, exploiting rich data structures in external languages. This by far is a better solution than implementing scripts that employ Unix greps or regular expressions.

VII. SUCCESS STORIES

Our UVM scoreboard architecture has been across numerous UVM/VMM projects. Typically we see such projects obtaining an approximate 15% code reduction compared to creating the scoreboard from scratch using the empty uvm_scoreboard class. Scoreboard setup, configuration and validation can be done in less than a day, even for complex designs, offering easy ramp-up for engineers new to UVM and the act of scoreboarding. Furthermore, experienced engineers easily pick up and extend test benches created using the scoreboard library, as the scoreboard look and feel is same across applications and engineers. Out of the box, engineers benefit from an inherent top performing scoreboard with very good debug capabilities, prepared for hooking up to external interfaces.

VIII. IMPROVEMENTS

Since first published [4], the underlying implementation of the UVM scoreboard has undergone some changes in order to improve the usability and tool compatibility with the three big simulator vendors. In general three major changes have been done:

- Changes in the class hierarchy. The structure of the class hierarchy has not been changed but some changes to the class names and the inheritance hierarchy has been done. All classes has been renamed from cl_scb_uvm_* to cl_syoscb_* to shorten the file/class names a bit. In the first release, almost all classes where derived from uvm_component. However, this was not optimal, since then they were a part of the UVM phasing framework which is not needed for classes which are purely internal to the UVM

scoreboard. Most classes are now derived from uvm_object (to keep them within the UVM framework). Only cl_syoscb_compare and cl_syoscb_queue are derived from uvm_component since they need to be a part of the phasing, in turn being able to raise objections and/or do error reporting. One could argue that UVM lacks a "uvm_datastructure" class which is derived from uvm_object and could be used for implementing generic data structures while keeping them within the UVM framework.

- Different methods for enforcing APIs. The previous release used abstract classes (specified with the keyword "virtual class" in SystemVerilog) with pure virtual classes in the cl_syoscb_queue and cl_syoscb_queue_iterator_base classes. According to the SystemVerilog LRM [5] section 8.21 then it is legal to extend a non-abstract class into an abstract class as long as: "An object of an abstract class shall not be constructed directly. Its constructor may only be called indirectly". Thus, enforcing APIs via virtual classes with pure virtual functions should work. However, this is not the case due to UVMs factory implementation.

  The problem appears when using the UVM macros in a class derived from uvm_object. The macro generated code for the <class>::create_instance(…) method will call the constructor directly, deep within the UVM factory code. This is legal as long as the class is not an abstract class as explained above. In the previous implementation of the UVM scoreboard the cl_syoscb_queue class was exactly an abstract class with a constructor which was called directly due to the UVM factory macros. Thus, this code was incorrect according to the SystemVerilog LRM. Some of the simulator vendors tools caught this at compile time while others did not. Hence, we have now changed the way we enforce APIs to avoid using abstract classes and solely use virtual methods which issues a uvm_fatal if not overridden.

  The optimal solution would be to make the UVM factory capable of handling abstract classes, by adding an option to the UVM macros explicitly stating that this is an abstract class and should be handled differently. Another way of enforcing an API could be to use the newly added SystemVerilog interfaces classes. This solution was not chosen since we wanted a stable implementation with stable simulator support. Currently, we are expecting too many simulator problems with this new feature.

- Minor changes to obtain simulator compatibility. The UVM scoreboard has been tested on all of the major simulators. Besides the problem explained above with abstract classes only minor things have been changed. In general, references more than one object deep cannot be handled by all simulators, e.g. to an obj.method().member reference. Thus, minor changes to avoid these situations have been implemented. The UVM scoreboard has been tested with the following simulators:

  o Synopsys VCS® version J-2014.12

  o Mentor Questa® Advanced Simulator version 10.3c

  o Cadence® Incisive® version 14.10.014

IX.   GENERAL AVAILABIILITY

Our UVM Scoreboard architecture has been released for general availability, and can be downloaded from following web resources:

- Accellera UVM Forum

  o http://forums.accellera.org/files/file/119-versatile-uvm-scoreboard/

- SyoSil homepage

  o http://syosil.com/index.php?pageid=33

The release features the UVM Scoreboard base classes, examples as well as 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

X.   FUTURE EXTENSIONS

SyoSil is committed to continue the development of the UVM Scoreboard. We currently plan to include following extensions before summer 2015:

- Additional implementations of compare methods, including examples of how to employ special rules taking the contents of the UVM sequence items into consideration.

- Additional queue implementations, optimized with better locators

- More configuration "knobs"

- A general mechanism for communicating side effects from the reference model to the scoreboard

- Richer set of examples, covering the non-UVM connectivity

- Even better debug aiding mechanisms

- Work with the simulator vendors to correct the bugs found in connection with abstract classes and direct invocation of the class constructor

XI.   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.

[1]   Accellera Forum, UVM Resources, http://forums.accellera.org/files/category/3-uvm/

[2]   Mentor Graphics, UVM Connect Library, https://verificationacademy.com/topics/verification-methodology/uvm-connect

[3]   Accellera Standards, SCE-MI (Standard Co-Emulation Modeling Interface), http://www.accellera.org/downloads/standards/sce-mi

[4]   Versatile UVM Scoreboarding, Andersen, Jensen & Steffensen, DVCon Europe 2014

[5]   "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language." IEEE Std 1800-2012, 2012.