Standardizing Verification IP Reuse by Introducing SystemVerilog Verification Components

Jacob Andersen
Peter Jensen
Stig Kofoed

SyoSil
Systems on Silicon

Himmelev Bygade 53, 4000 Roskilde, Denmark
www.syosil.com
{jacob, peter, stig}@syosil.com

ABSTRACT

This paper introduces an industry-proven, standardized way of writing VMM compliant Verification IP, namely the concept of SystemVerilog Verification Components (SVVCs). The SVVC concept standardizes how VIP offering support for both directed testing and constrained random verification is built across protocols, and offers a common look and feel. The SVVC concept offers a well documented approach of how to verify the VIP components in a stand-alone context. This approach decouples the VIP development from any RTL design development effort, and ensures that the SVVCs are fully verified before being employed in an RTL test bench. This paper also describes how VMM compliant test benches are rapidly composed based on reusing SVVCs and other generic verification components, such as scoreboards and reference models. Furthermore, on the top of such SVVC based test benches, we show how directed and constrained random test cases access the SVVCs.

Table of Contents

Table of Figures

1.0 Introduction

To address the increasing challenges met when performing functional verification of state-of-the-art digital integrated devices, methodologies such as VMM [2] combined with the HDVL SystemVerilog [1] is currently enabling an evolution towards true coverage-driven, constrained random verification.

While offering numerous new tools for the verification engineer toolbox, VMM and SystemVerilog also challenges the engineer with a steep learning curve and an increased amount of complexity in the areas of languages, tools and methodologies, something that initially threatens to drastically lower the engineer productivity. To address these challenges, it is of great importance to be able to employ a high degree of reusability of verification components, a concept also known as Verification IP (VIP).

Today the VMM methodology offers a rich set of guidelines for ensuring that VMM based VIP is reusable, e.g. with regard to concepts and connectivity. Still, we have seen that independently built VMM compliant VIP turns out to be hard to reuse. Every independently VMM trained engineer writes VIP in her or his own way.

This paper introduces an industry-proven, standardized way of writing VMM compliant Verification IP, namely the concept of SystemVerilog Verification Components (SVVCs). The SVVC concept standardizes how VIP offering support for both directed testing and constrained random verification is built across protocols, and offers a common look and feel. Even for complex industry-standard protocols, SVVC authors know exactly what to write, and SVVC users know exactly what features to expect and how to invoke them. This gives a superior engineering productivity, which in turn leads to finding more RTL design bugs, as the verification engineer ultimately is able to focus on the device, rather than spending time on debugging a complex hard-to-understand verification environment.

The SVVC concept also offers a well documented approach of how to verify the VIP components in a stand-alone context. This approach decouples the VIP development from any RTL design development effort, and ensures that the SVVCs are fully verified before being employed in an RTL test bench. This eliminates the scenario of having functional errors in the VIP that cloak errors in the RTL, which greatly enhances verification engineer productivity and the general quality of the verification process.

Beyond describing how to create and verify the SVVCs, this paper outlines how VMM compliant test benches are rapidly composed based on reusing SVVCs and other generic verification components, such as scoreboards and reference models. Furthermore, on the top of such SVVC based test benches, we show how directed and constrained random test cases access the SVVCs.

2.0 The SyoSil Layered Approach to VMM Verification

All state-of-the-art verification environments are built in layers. VMM enforces such a layered approach in order to promote maximum reuse and scalability of VIP, test bench components and

test cases. The SyoSil framework for developing VIP, test benches and test cases extends the VMM layered approach as shown below.

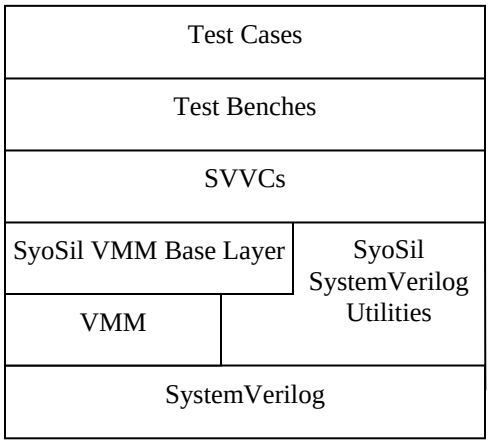

Figure 1 – Layered Approach Extending VMM

The *SyoSil SystemVerilog Utilities* consist of multiple resources streamlining and accelerating the creation of the upper layers. To mention a few, the utilities contain:

- Low level libraries (generic messaging services, string handling services).
- A generic scoreboard architecture.
- A base class framework for modelling reference models and configuration state registers.

The *SyoSil VMM Base Layer* extends most of the VMM base classes with additional functionality to support the SVVC approach. The layer encapsulates all code that is shared across different SVVCs and test benches, and keeps the amount of code here to the application specific minimum. Examples of such shared code are:

- Simulation switches (reference model and scoreboard control).
- Registration and maintenance of SVVCs in test bench.
- Control of information output level throughout SVVCs and the test bench.

Note that all VMM compliant code can run together with this base layer, and any VMM documentation applies as well.

Based on the above mentioned layers SVVCs, test benches and test cases are created while utilizing all the code present in the lower layers. Similarly to using the bare VMM base classes, the layered approach used for the SVVC framework simplifies the challenge of creating a running system for the end user.

3.0 Limiting the Use of the VMM Feature Set

VMM is a rich methodology which offers many different both simple and complex concepts and constructs to develop VIP and test benches. This wealth of opportunities is what makes modern verification engineering difficult – simply choosing the best way of implementing a certain solution. This leads to diversity as each engineer chooses her or his own style for implementing VIP, test benches, test cases, functional coverage, etc., which clearly is a disadvantage when trying to create uniform and reusable verification components.

To address this challenge, we have chosen to limit the number of VMM features used for certain tasks, and furthermore put forward internal standardization guidelines in order to "keep things simple". Below, a selected number of these guidelines are presented.

3.1 Test Bench Structure Using VMM Channels Only

VMM suggests a basic test bench infrastructure, in which channels connect transactors, e.g. generators and Bus Functional Models (BFMs). Using these channels, transactions spend their lifetime by travelling from their point of creation till their destruction, while their presence causes various actions to happen, e.g. causing the BFM to send and receive stimuli from the Design Under Verification (DUV).

The infrastructure created using channels can easily be depicted in a test bench specification, and is simple to understand for people with a hardware background. However, when it comes to hooking up functionality such as reference models, scoreboards and units sampling transaction coverage, VMM recommends that hook up is made using call-backs [2]. Using call-backs is a well known powerful software technique for run-time modification of code, but when put into a hardware verification perspective, verification engineers find it difficult to overview test benches with heavy use of call-back mechanisms.

To preserve a clear notion of test bench structure, we propose to avoid using call-backs for composing test bench structure. Rather, reference models, scoreboards and coverage units are created as true VMM transactors capable of hooking up to channels connecting these transactors to the other test bench components. This yields a more clear and well defined test bench infrastructure, which still is run-time constructed and modifiable, e.g. based on randomization of the DUV structure and capabilities which can be controlled by parameters.

The use of call-backs is in general not prohibited. A call-back remains a powerful mechanism for modifying e.g. BFM behaviour (dropping or delaying transactions), depending on DUV and test bench dynamics. But the use of call-backs should be left for implementation of such exceptions, and not be used for implementing test bench structure.

3.2 Cross Referencing Transactions from Multiple Transactors

The channel based infrastructure of a test bench in fact does not move transactions around between transactors. Rather, references to transactions are passed around. This allows a transactor – even though it already passed a transaction on to the transactor next in chain – to keep a reference to the transaction. Thereby multiple structural elements may have access to the

same transaction at the same time, and potentially is capable of modifying the transaction simultaneously. VMM does not prevent such hazardous actions from happening. For some simple uses, VMM actually endorses such activity [2].

We have chosen to employ a strict policy regarding cross referencing transactions from multiple transactors:

1. A transaction should only be referenced by one single transactor or channel at a time.
   - Once a transactor performs a *put* or a *sneak* of a transaction on a channel, it should no longer reference that object.
   - To get a transaction from a channel, a transactor should employ the channel *get* method and defer from using the *peek* and *tee* methods.
   - This implies that advanced channel functionality such as the *active slot* and numerous other more advanced VMM channel features (e.g. VMM notifications) are left unused in this framework.
2. A VMM broadcaster sending copies of transactions to multiple consumers should employ a true deep copy method and defer from only copying transaction references.
   - This prevents that e.g. a reference model by mistake modifies a transaction that also goes into the scoreboard, which would be hazardous to the verification safety.

Although copying data is less efficient we have not yet observed any substantial performance impact. If performance becomes an issue the broadcaster can be configured to copy references instead of deep copying the objects, as explained in [2].

4.0 Extending VMM: Hierarchical Test Benches

Traditionally a VMM test bench is flat in structure (the next generation of VMM introduces hierarchical test benches through the *vmm_subenv* class). Basically it consists of a *vmm_env* [2] encapsulating numerous transactors, channels and alike. This also implies that reusable VIP normally consists of an ungrouped collection of:

- Data classes.
- VIP configuration mechanisms.
- Transaction generators (transactors).
- BFMs and monitors (transactors).
- Interfaces and assertions checkers.

For each DUV interface, these numerous VIP subcomponents must be declared, constructed and controlled inside the top level test bench scope, namely the *vmm_env*. This leads to a very crowded and difficult to understand top level structure, especially when the DUV has many interface instances.

To address this, we have standardized how hierarchical test benches are built from multiple *vmm_env*'s. This allows grouping all the mentioned VIP sub components into a single *vmm_env* environment, which makes it extremely easy to reuse. It allows easy declaration, construction,

hook up and invocation of a VIP component. Also, the VIP environment can be further modified and customized by employing class inheritance of the environment.

The SyoSil VMM base layer offers an extension of the *vmm_env* class, called *vmm_env_syo*. Using this extension automates the calling of each sub environment (in this context SVVC) by simple registration of the sub environment in a data structure of the parent environment. For each of the *vmm_env_syo* sub steps (*gen_cfg()*, *build()*, etc), the parent environment calls the same method for each of its sub environments.

5.0 The SVVC

5.1 Introduction

The SystemVerilog Verification Component (SVVC) concept was developed by SyoSil to standardize how VIP is built across protocols. The SVVC concept is independent of the SystemVerilog methodology used (here VMM), but any actual SVVC implementation is methodology specific.

Using the SVVC concept instead of writing VIP in any arbitrary form or structure offers following advantages:

- Proven support for constrained random verification and directed testing.
- A common look and feel: SVVC authors know exactly what to write, and SVVC users know exactly what features to expect and how to invoke them.
- Stand-alone verification of the SVVC, which enables developing the SVVC independently of the availability and state of any RTL block using the same protocol.
- Easy and standardized reuse integration of multiple SVVCs in large test benches.

Protocols are by nature quite different. Some are simple master/slave protocols, whereas some are highly pipelined multi-point protocols with independent address and data phases and using tagged transfers. Experience from industry applications has shown that the SVVC concept addresses both simple and complex protocols, but special protocols require adaptations to the basic SVVC concept in rare cases.

5.2 Overview of the SVVC Structure

An SVVC is a SystemVerilog class based object connecting to a DUV RTL port in the form of a SystemVerilog RTL interface by using virtual interface hook up. The SVVC will drive and respond to the physical protocol, based on the high level transactions generated internally in the SVVC or requested externally by the test bench. In its most simple form, this may be an SVVC method call (e.g. *send_transaction()*) invoked by the test bench which creates a bus transaction on the physical protocol, but may also be complex random scenarios conducted by the SVVC. The SVVC is driven by generators and transactors. A generator produces transactions, whereas a transactor also acts as a consumer.

The SVVC itself is a sub environment derived from *vmm_env_syo*, in which all the objects that constitute the SVVC are instantiated. The basic SVVC structure is depicted in the figure below.

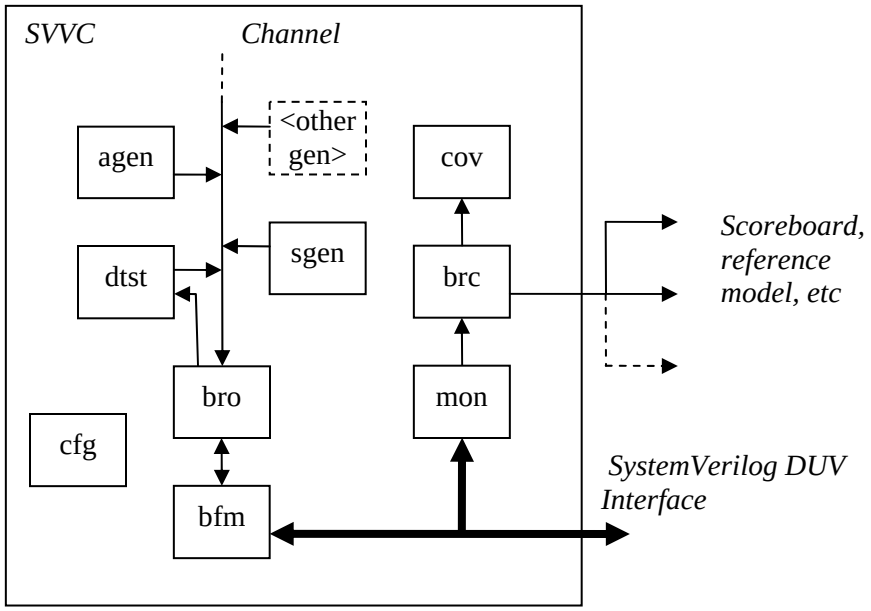

Figure 2 – SVVC Structure

The following objects exist inside a VMM based SVVC:

Object Name	Class Name	Description
agen	cl_*svvc*_atomic_gen	Atomic constrained random generator dispatching independent randomly generated transactions to the broker.
sgen	cl_*svvc*_scenario_gen	Scenario constrained random generator dispatching randomly generated sequences of transactions to the broker.
dtst	cl_*svvc*_dtst	Directed test transactor dispatching and receiving direct test case transactions to/from the broker.
<other gen>	cl_*svvc*_<other gen>	Other customized generator(s) specific for this SVVC type.

bro	cl_*svvc*_broker	Broker exchanging information between transaction producers/consumers and the *bfm*. Note that additional transaction producers/consumers beyond the *agen*, *sgen* and *dtst* units may be registered on the broker.
bfm	cl_*svvc*_bfm	Bus Functional Model driving and responding to the DUV interface according to the protocol. The *bfm* typically implements a requestor (master) side and a responder (slave) side. Implementing monitor/observer functionality in the *bfm* is mandatory.
mon	cl_*svvc*_bfm	Monitor sampling all transactions on the DUV interface. Often the same class as the *bfm*, but configured to act as a monitor (passive).
brc	vmm_broadcast_syo	VMM broadcaster distributing all transactions from the monitor to all consumers.
cov	cl_*svvc*_cover	Transactor collecting functional coverage of the transactions collected by the monitor, received via the *brc*.
cfg	cl_*svvc*_cfg	Configuration class for SVVC, setting e.g. bus latencies on the DUV interface.

The following other classes are used in or are part of the SVVC:

Class Name	Description
cl_*svvc*_data	Data class capturing one transaction for the protocol. VMM macro expansion generates the classes cl_*svvc*_channel, cl_*svvc*_atomic_gen and cl_*svvc*_scenario_gen based on this data class.
cl_*svvc*_data_channel	Channel class able to transport cl_*svvc*_data or a descendant hereof.
cl_*svvc*_env	The SVVC itself (sub environment)

Readers familiar with VMM will recognize basic VMM concepts such as the data class, the atomic/scenario generator classes, the channel class and various extensions of the VMM

transactor base class. The process of creating the protocol specific extensions of these VMM base classes is well described in [2], and will not be discussed further in this paper.

5.3 The Bus Functional Model (Class cl_*svvc*_bfm)

The *bfm* is the link between the higher level abstraction (namely the data class) and the DUV interface. The BFM class must be able to act on the physical pin level interface in the different configurations described in the protocol specification, typically to act both as a master and a slave. Furthermore, all BFMs shall implement a monitor BFM. One and same transactor shall be configurable such that it can act as any of the mentioned types.

Three different categories of BFMs exist: Requestors, responders and passive transactors. It is important to determine what category each BFM type falls into. Note that all BFM categories use same transaction type (data class), but various fields may not be used for e.g. a response transaction. Often the three BFM categories are implemented in a single class to enable code reuse and easier maintenance, but they could also be implemented as independent classes.

For some protocols the BFM must be able to handle overlapping transactions. Below, the behaviour for a single transaction is outlined for each of the three BFM categories.

5.3.1 Requestor BFM Type

A requestor *bfm* has an initiating nature (master) and only starts sending on the DUV interface if a transaction is sent from the *bro* (originates e.g. from the *agen* or the *dtst*). Activity cannot be initiated by the DUV interface. For each transaction, a requestor *bfm* goes through the following steps:

1. The *bfm* waits until the *bro* sends a transaction.
2. The requestor *bfm* drives the transaction on the DUV interface, and awaits any possible response. This process may take several clock cycles and/or sequence of activities on the DUV interface.
3. When the response has been received, the *bfm* aggregates the complete transaction, and sends this back to the *bro*.
4. For some protocols, exchange of multiple sub transactions between the *bro* and the *bfm* might be required to complete a single transaction.
5. The sequence is restarted at #1.

5.3.2 Responder BFM Type

A responder *bfm* has an awaiting nature (slave) and awaits activity on the DUV interface before communicating any transactions with the *bro*. Activity cannot be initiated by the *bro*. For each transaction, a responder *bfm* goes through following steps:

1. The *bfm* awaits a valid request on the DUV interface.
2. A received request is packaged in a transaction, and sent to the *bro*, which is responsible for retrieving an answer for that request, e.g. from the *agen* or the *dtst*.

3. The *bro* must send a response transaction back in same time step, otherwise the *bfm* will be unavailable to serve the request and will probably break the physical protocol, or in the best case cause wait states to be inserted on the protocol.
4. When the *bfm* has received a response transaction from the *bro*, the physical transaction can be completed.
5. The *bfm* aggregates the complete transaction, and sends this back to the *bro*.
6. For some protocols, exchange of multiple sub transactions between the *bro* and the *bfm* might be required to complete a single transaction.
7. The sequence is restarted at #1.

5.3.3 Passive BFM Type

A passive *bfm* type has an awaiting nature, and furthermore does not interact actively with the physical protocol – it only listens (monitor). The role of the monitor is to listen for a transaction on the protocol, and forward this transaction. For each transaction, a passive *bfm* goes through following steps:

1. The *bfm* awaits a valid transaction on the DUV interface.
2. The *bfm* aggregates the complete transaction, and sends this out to the *brc*.
3. The sequence is restarted at #1.

Scoreboarding, coverage, feeding reference models and alike shall be done based on actual protocol traffic. This is done by using a monitor *bfm*, and distributing the data packages from the monitor to these receivers.

Note that an idle DUV interface does not carry any transactions. A monitor listening to such an idle interface will not produce any transactions until real protocol activity happens.

5.3.4 BFM Example

The following example shows how the three different BFM categories easily can be implemented into a single class. The protocol used in the example has a MASTER side (requestor type), SLAVE side (responder type) and a MONITOR (passive type). The control of the pins on the DUV interface for each of the three types is implemented in the three methods: *do_master()*, *do_slave()* and *do_monitor()*. Furthermore, the example shows how constraints and coverage for protocol specific delays are implemented.

```
class cl_mysvvc_bfm extends vmm_xactor_syo;
  // Transaction input channel
  cl_mysvvc_data_channel in_chan;

  // BFM kind
  typedef enum {NOTSET, MASTER, SLAVE, MONITOR} tp_kind;
  tp_kind kind;

  // Protocol delays to be randomized
  rand int master_delay;   // Protocol master delay

  // Constrain the randomization of the protocol delays
  constraint co_master_delay {
    ...
  }

  // Cover the randomized delays
  covergroup cg_master_delays;
    cp_master_delay : coverpoint master_delay {...}
    ...
  endgroup

  ...
  // VMM compliant API
  extern function new (cl_mysvvc_cfg cfg = null,
                       virtual ifduv vi,
                       tp_kind kind = cl_mysvvc_bfm::NOTSET);

  extern virtual task automatic main();
  extern virtual task automatic reset();

  extern protected virtual task automatic do_master (
    ref cl_mysvvc_data tr);

  extern protected virtual task automatic do_slave  (
    ref cl_mysvvc_data tr);

  extern protected virtual task automatic do_monitor(
    ref cl_mysvvc_data tr);

  // pre_randomize and post_randomize allows modifying the
  // randomization result
  extern function void pre_randomize();
  extern function void post_randomize();

endclass
```

Figure 3 – *bfm* Class Definition Example

The following example shows the main loop of the BFM. Note how the protocol delays are randomized by invoking *this.randomize()* and how the protocol delay coverage is sampled. At the very end of the loop the pin control method that corresponds to the BFM type is invoked on the transaction with randomized delays.

```
task automatic cl_mysvvc_bfm::main();
  cl_mysvvc_data tr;
  bit drop;

  ...

  // Main loop to drive the Protocol bus
  while (1) begin
    if (this.kind == cl_mysvvc_bfm::MASTER) begin
      // BFMs of REQUESTOR type
      // Get a transaction from the input channel
      this.in_chan.get(tr);

      // Randomize the BFM delays
      if (!this.randomize()) `vmm_error(log, "...");
    end else if (this.kind == cl_mysvvc_bfm::SLAVE) begin
      // BFMs of RESPONDER type
      ...
    end else begin
      // BFMs of PASSIVE type
      ...
    end

    ...

    // Do coverage of protocol delays
    case (kind)
      cl_mysvvc_bfm::MASTER : cg_master_delays.sample();
      ...
    endcase

    // Process the transaction
    case (this.kind)
      cl_mysvvc_bfm::MASTER :  do_master(tr);
      cl_mysvvc_bfm::SLAVE  :  do_slave(tr);
      cl_mysvvc_bfm::MONITOR:  do_monitor(tr);
      default               :  `vmm_error(log, "Unknown BFM kind");
    endcase
    ...
  end
endtask
```

Figure 4 – *bfm main()* Method Example

5.4 The Broker (Class cl_*svvc*_broker)

The motivation for introducing the *bro* in the SVVC framework is to standardize how to exchange transactions between the *bfm* and the *agen*, *dtst* and other producers.

Generators simply produce transactions on a channel and do not expect a response. The *agen* generator is an example of this. On the other hand, transactors produce transactions on a channel, and expect to receive a response from a channel to operate properly. The *dtst* is an example of this. The *bro* has been introduced to shield the typically rather complex *bfm* implementation from these details.

Similarly to the *bfm*, the *bro* can be either of requestor or responder type. No passive type can exist. A requestor type *bro* awaits an initiative from a producer before sending a transaction to the *bfm*, whereas a responder type *bro* awaits an initiative from the *bfm* before attempting to send a transaction to a producer. Obviously, a requestor type *bro* is used together with a requestor type *bfm*, and a responder type *bro* together with a responder type *bfm*.

The broker concept is currently limited to only having a single client with an input channel (e.g. a *dtst*) registered on a responder type broker. No such limitation exists on a requestor type broker.

A standard *bro* will work using the algorithms outlined below. Note that any number of generators can be registered on the *bro*.

5.4.1 Requestor Type

1. When a producer sends a request transaction to the *bro*, the *bro* forwards the transaction to the *bfm*. The *bro* also keeps track of the producer identification number which the transaction was tagged with.
2. When the *bfm* returns the transaction after processing it, and possibly updating the transaction with the protocol response, the *bro* sends back the transaction to the producer originally creating the transaction (this is the case for a typical *dtst*). If the producer return channel does not exist, the return transaction is simply discarded by the *bro* (this is the case for an *agen*).
3. Continue at #1. Note that the steps #1 and #2 can run in parallel, such that the *bfm* can process multiple transactions in parallel.

5.4.2 Responder Type

1. When the *bfm* sends a request transaction, the *bro* tries to deliver it to a producer:
    a. If only a generator with no input channel (e.g. an *agen*) is present and has a response transaction ready, then that response transaction is sent to the *bfm* immediately and the original *bfm* request transaction is discarded.
    b. Otherwise, if a transactor with an input channel (e.g. a *dtst*) is present, then the request transaction from the *bfm* is sent to the transactor. The *bro* will then await a response transaction from that transactor, which has to arrive in same time step to avoid breaking the protocol, or in the best case cause protocol wait states to be

inserted. When the transactor sends a response transaction, this is forwarded to the *bfm*. This mechanism allows a transactor to produce a response transaction based on the request transaction from the *bfm*.

2. If both steps #1.a and #1.b failed to service the *bfm* request, a fatal error is reported, as the protocol breaks.
3. Continue at #1.

5.5 The Direct Test Transactor (Class cl_*svvc*_dtst)

The direct test transactor (*dtst*) is responsible for offering an API that allows a directed test case to create transactions to be sent on an interface (requestor type BFM), as well as respond to requests on an interface (responder type BFM). As protocols vary much in nature, the offered API will be very protocol specific, beyond the fact that such method calls either belongs to a requestor or responder BFM type.

A *dtst* on a requestor BFM side will remain silent unless the test bench or a test case invokes a method which creates a transaction. Such method calls can be both blocking and non-blocking, depending on whether they wait and return the response coming from the responder side.

A *dtst* on a responder BFM side will typically have blocking method calls, which returns once a transaction arrives on the DUV interface. The *dtst* service agent (test bench or a test case) should then immediately call a method delivering the response, to avoid breaking the physical protocol.

The arguments to *dtst* method calls are typically simple data types and not data class objects. This allows a directed test case writer to carry out his work without even knowing the existence of data class objects. The *dtst* method calls basically come in two different flavours:

1. A simple call either sending a transaction or waiting for a transaction. This simple call basically just sends/gets a transaction and assembles/disassembles a transaction of the data class type.
2. A more complex call sending a transaction and also awaiting a response before returning. Such functionality requires mechanisms for keeping track of ongoing and returning transactions, for instance based on the transaction ID numbers carried by each transaction or a tagging mechanism on the DUV interface.

The following example gives a simple implementation of a *dtst* object providing a simple API, which consists of a single method, namely the *write()* method. The *write()* method takes an address and data as input arguments. It creates a data object containing the write transaction, and sends it to the broker through the `out_chan`.

```
class cl_mysvvc_dtst extends vmm_xactor_syo;
  // Transaction channels
  cl_mysvvc_data_channel out_chan, ...;

  // Factory object - can be used to further constrain the objects
  // to be sent on the out_chan towards the BFM
  cl_mysvvc_data randomized_obj;

  // Configure whether this transactor is blocking
  // Can be changed runtime
  bit blocking = 0;

  // VMM compliant API
  extern function new (cl_mysvvc_cfg cfg = null, ...);

  extern virtual task main();

  // Generic object creation method
  task automatic send_obj(cl_mysvvc_data obj);
    cl_mysvvc_data o;
    $cast(o, obj.copy());
    if (blocking) begin
      out_chan.put(o);
    end else begin
      out_chan.sneak(o);
    end
  endtask

  // MASTER : Specific object methods
  task automatic write (logic [31:0] i_adr, logic [31:0] i_dat);
    int status;
    status = randomized_obj.randomize() with
                           {     op   == cl_mysvvc_data::WRITE;
                                 adr  == i_adr;
                                 dat  == i_dat;
                           };
    if (!status) `vmm_error(log, "Randomization failed!");
    send_obj(randomized_obj, ...);
  endtask
endclass
```

Figure 5 – *dtst* Class Definition Example

5.6 The Stimuli and Response Coverage (Class cl_*svvc*_cover)

The *cov* SVVC member is designed to collect functional coverage of the transactions sampled by the monitor type BFM. This will check if all possible transactions have been exercised on each interface. The implementation of this is highly protocol specific.

5.7 The Configuration (Class cl_*svvc*_cfg)

The *cfg* SVVC member contains a number of data members, which allow configuration of:

- General SVVC settings.
- Shared settings between SVVC parts.
- SVVC component specific settings.

For instance, component specific settings may be constraints for maximum bus latencies used when randomizing the bus protocol delays in the BFM. Such settings will be used both for constraints and coverage goals.

5.8 The SVVC Top Level (Class cl_*svvc*_env)

The SVVC top level is based on the hierarchical test bench approach described in section 4.0. This approach makes it very simple to instantiate an SVVC in the test bench, and connects it to the respective DUV interface. The SVVCs sub components (*agen*, *bfm*, etc) should not be directly instantiated in a test bench.

The example below shows an implementation of the SVVC top level. Remember that it is a sub environment derived from *vmm_env_syo*, in which all the objects that constitute the SVVC are instantiated.

```
class cl_mysvvc_env extends vmm_env_syo;
  // Protocol DUT interface
  virtual ifduv if1;

  // Atomic generator
  cl_mysvvc_data_atomic_gen agen;

  // Transactor for directed testing
  cl_mysvvc_dtst dtst;

  // Broker for BFM arbitration between dtst and agen
  cl_mysvvc_broker bro;

  // BFM for protocol interface
  cl_mysvvc_bfm bfm;

  // Monitor BFM for protocol interface
  cl_mysvvc_bfm mon;

  // Broadcast for scoreboard/referencemodel/coverage
  vmm_broadcast_syo brc;

  // Coverage transactors
  cl_mysvvc_cover cov;

  // Configuration
  cl_mysvvc_cfg cfg;

  // The BFM type acting/reacting on the protocol
  // interface
  cl_mysvvc_bfm::tp_kind bfm_kind;
  ...
  // Constructor
  extern function new(..., virtual ifduv if1,
                      cl_mysvvc_bfm::tp_kind bfm_kind, ...);

  // The 9 VMM environment steps
  extern virtual function void gen_cfg();
  extern virtual function void build();
  ...
  extern virtual task automatic report();
endclass: cl_mysvvc_env
```

Figure 6 – SVVC Class Definition Example

The test bench does not need to call the various *vmm_env_syo* step methods (*build()*, *start()*, etc) inside the SVVC. The test bench writer only needs to "register" the SVVC in the test bench environment, which automatically will call these methods in the SVVC as shown below.

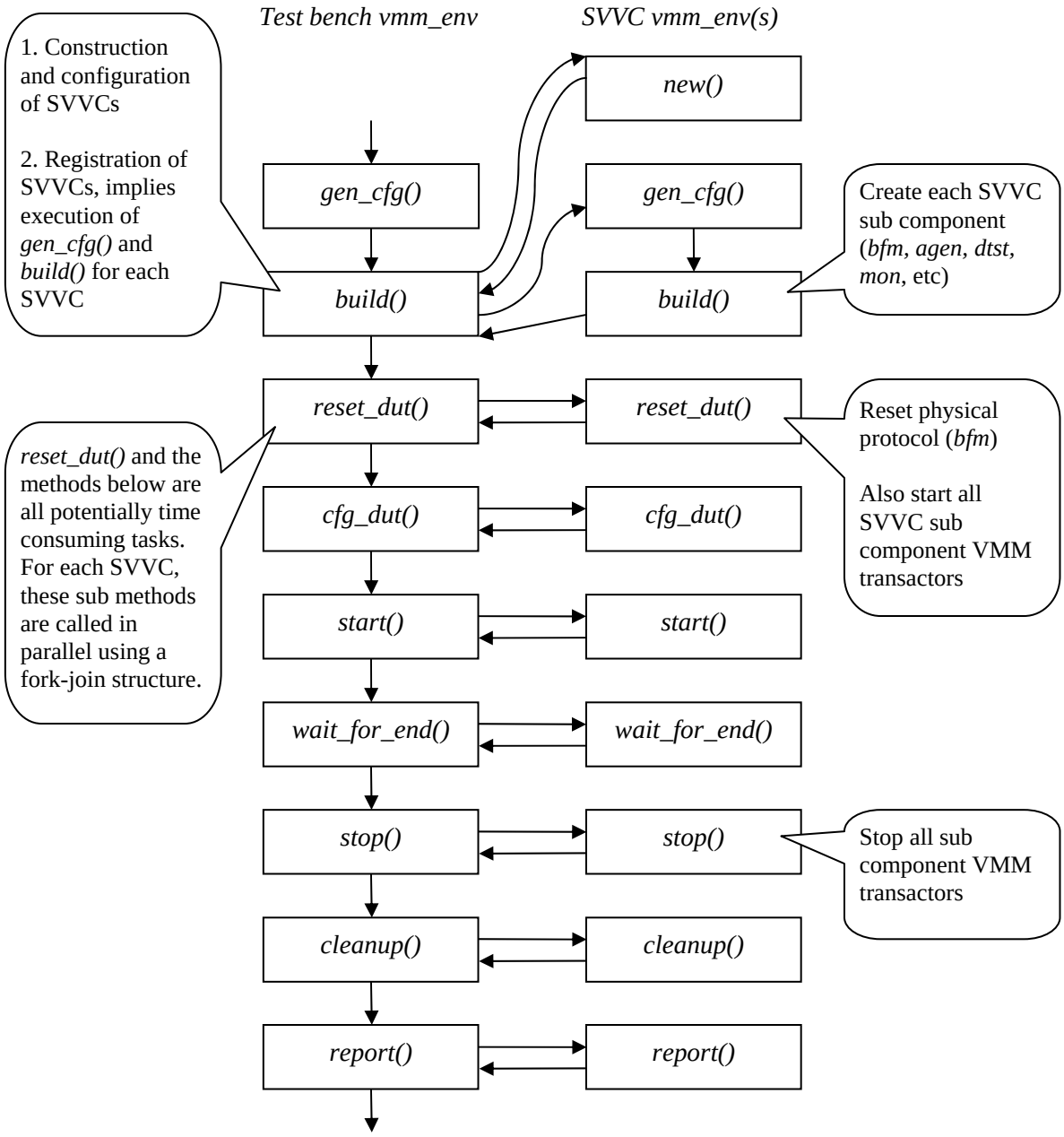

Figure 7 – SVVC Execution Sequence

Typically, the constructor is responsible for the creation of each object inside a container object, such as the SVVC environment, but in this case the creation of the *agen*, *bfm*, *bro* etc. is done in the *build()* method. Hence, the creation of these objects can depend on the configuration generated by the *gen_cfg()* method. The example below gives an example of such a *build()* method. Note how the *agen* and *dtst* are registered to the broker.

```
function void cl_mysvvc_env::build();
  // Call super's build
  super.build();

  // Create BFM and broker
  case (bfm_kind)
    cl_mysvvc_bfm::MASTER : begin
        bfm = new(cfg,..., dutif, cl_mysvvc_bfm::MASTER);
        bro = new(cfg,..., cl_mysvvc_broker::REQUESTOR);
      end
    cl_mysvvc_bfm::SLAVE  : ...
    default               : `vmm_fatal(log, "This MYSVVC env ...");
  endcase

  // Create atomic random generator
  agen = new(...);

  // Create transactor for directed testing
  dtst = new(cfg,...);

  // Do the hook up of broker and bfm
  bro.bfm_out_chan.connect(bfm.in_chan);

  ...
  // Register agen and dtst on broker
  // agen has to be first as it has no input channel. Remember the
  // broker only support a single transactor with in/out channels
  // (the dtst), and that has been registered last
  bro.register_transactor(agen.out_chan);
  bro.register_transactor(dtst.out_chan, dtst.in_chan);

  ...
endfunction
```

Figure 8 – SVVC *build()* Method

6.0 SVVC Verification

Too often, the work of a verification engineer developing VIP and test benches is started after the RTL design process has finished or at least reached a stable design state. As the complete verification task has grown to overshadow the RTL design task in both a matter of working hours but often also the complexity, it is important that the verification development process is decoupled from that of the RTL design. To avoid verification being the bottleneck, this process should be initiated as soon as design specifications are created and DUV interfaces have been documented.

Verification components (VIP and test benches) should – if possible – be reused from an existing code base. If components are to be engineered from scratch or simply updated with new features, a methodology of how to verify the verification components (sic!) is important.

The SVVC framework for creating VIP has a well defined methodology for how to verify verification components in a stand-alone fashion, offering several advantages:

- The SVVC is protocol compliant, and does not contain the same bugs as the RTL, which could happen if the SVVC is verified against a particular RTL implementation.
- The SVVC can be developed prior to RTL being available. This allows the SVVC and test bench to be developed before or simultaneously with the RTL, and to be ready when the RTL is in a state that allows verification to begin. This removes the SVVC and test bench creation from being in the critical path of the design and verification development cycle. A requirement is that the DUV interface is implemented before the SVVC development begins.

Based on the assumption that an SVVC always implements both sides of the protocol (e.g. both master and slave), the SVVC is mounted in a test bench as shown in the figure below. The SVVC is basically verified against itself by running both directed and random stimuli over a DUV interface connecting the two instances of the SVVCs. Protocol assertions serve as a reference model, checking that the BFM correctly implements the protocol. It is therefore very important that the assertions are implemented.

By running this setup, the SVVC developer initially is able to inspect if the SVVC runs the protocol correctly, e.g. if the requestor and the responder are able to communicate. A scoreboard will be attached, verifying that both the requestor BFM, the responder BFM and the passive BFM (monitor) have the same understanding of valid transactions on the interface.

Note how the scoreboard is attached directly to the BFMs. Normally a scoreboard only attaches to an SVVC via the *brc*, but since the BFM knows which transaction it has processed it can easily provide this information.

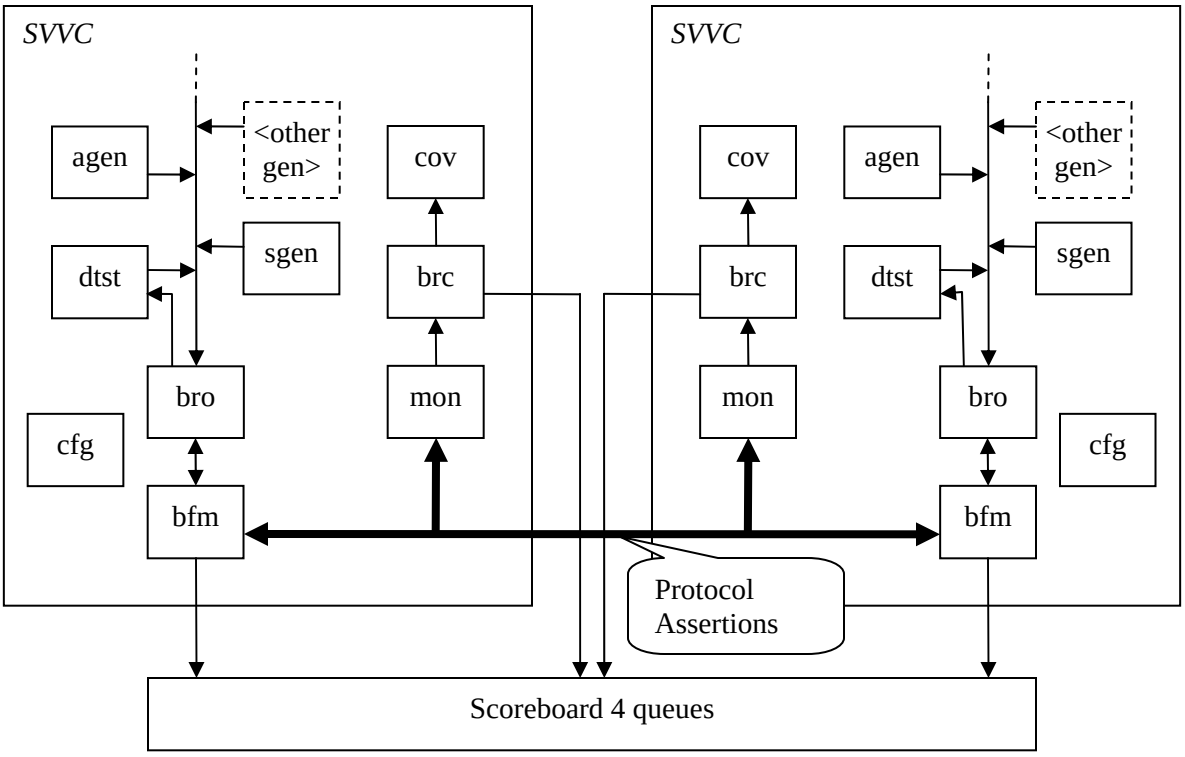

Figure 9 – SVVC Standalone Verification

The following criteria have to be met before an SVVC is considered verified and ready for integration into an RTL test bench:

1. Exercise all functionality of the direct test method calls using the *dtst.*
2. Run large number of random tests using the *agen* and *sgen.*
3. Obtain full coverage on the stimuli (in the *cov*).
4. Obtain full coverage on the protocol delays (in the *bfm*).
5. No scoreboard compare failure, all 4 queues from the requestor, responder and monitor should have same contents.
6. All assertions on the interface should hold and be covered.

Some protocols may have a special nature that requires that the SVVC verification methodology is carried out differently. But it should always be the goal to verify the SVVC standalone before integrating it into an RTL test bench.

7.0 Authoring VMM Test Benches Using SVVCs

7.1 Constructing, Registration and Accessing an SVVC

An SVVC is not usable as a standalone component. Rather, it is intended to be constructed inside a test bench, typically one SVVC per actual DUV interface. The figure below shows how SVVCs are used together with the test bench and the RTL.

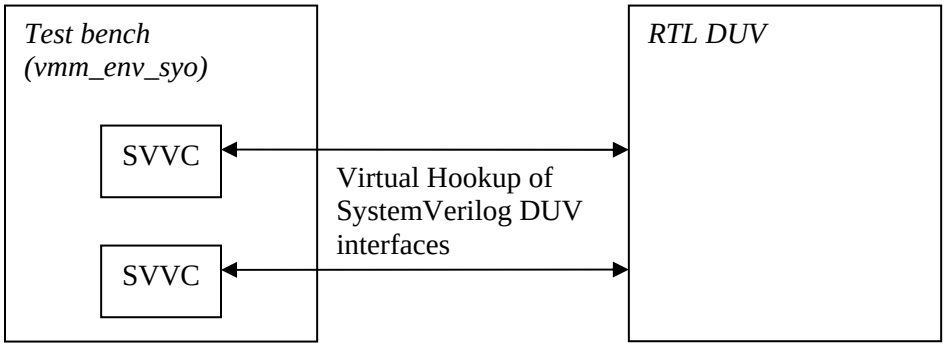

Figure 10 – SVVC Hook Up

An SVVC is a dynamic, run-time created component, not instantiated in a traditional HDL fashion. Therefore, no static hook up is performed, but still the SVVC has to access an RTL component, namely a SystemVerilog DUV interface. This is done by using the virtual interface hook up method, where the SVVC via its constructor is passed a pointer to an DUV interface. The SVVC then drives and samples that same DUV interface by reference throughout the simulation. The reference is typically not changed to be a different DUV interface instance in the middle of a simulation. However, the SVVC concept offers the opportunity for performing run-time changes to the standard SVVC configuration, both using class inheritance and call-back run-time code replacement. Moreover, an SVVC can be disconnected from the DUV interface during simulation, and be replaced by another SVVC taking over its task. This is obviously relevant for run-time configurable DUV interfaces.

The code below shows an example of declaring, constructing and registration of an SVVC inside a *vmm_env_syo* class, which is inherited from the *vmm_env* class in order to add functionality supporting the easy registration and run-time management of SVVCs. Remember, once the SVVC has been registered, all its subtasks are run automatically in sync with the top environment (here *cl_myenv*).

```
class cl_myenv extends vmm_env_syo;

  // Declare SVVC:
  cl_mysvvc_env if1_svvc;

  function void build();
    super.build();
    // Construct SVVC:
    // Args: instance_name, virtual_if, vmm_log, config_params
    if1_svvc = new("if1", if1, log, …);
    // Register SVVC
    register_localenv(if1_svvc);
    // Additional configuration of SVVC
    if1_svvc.cfg.someoption = …;
    …
  endfunction

  …
endclass
```

Figure 11 – SVVC Construction and Hook Up

7.2 Hook Up of Scoreboards and Reference Models

An SVVC also hooks up to external scoreboards and reference models. This is done by performing run-time registration of the channels feeding these external components on the *brc* inside the SVVC. All transactions sampled from the interface by the monitor are then automatically sent to the external components. The code below shows how easily scoreboards hook up to the SVVC. The same approach is used in an analogous fashion for reference models.

```
class cl_myenv extends vmm_env_syo;
  …

  // Declare the scoreboard
  cl_scb scb;

  function void build();
    …
    // Construct, hook up and start scoreboard
    scb = new(tbconfig);
    if1_svvc.brc.new_output(scb.rtl);
    scb.start_xactor();
    …
  endfunction

  …
endclass
```

Figure 12 – Hooking Up Scoreboard to SVVC

8.0 Authoring Test Cases

For a test bench (or test case) to control what transactions a SVVC sends and receives on a DUV interface, it should access the *agen*, *dtst* and similar objects inside the SVVC. All objects inside an SVVC that constitute the API to the test case are deliberately set to be public, so that objects and methods inside these SVVC subcomponents can be accessed directly. This provides great flexibility when writing test cases. For instance, to initialize a random sequence, the test bench may call the *start_xactor()* method of the agen. This use of public SVVC objects opposes normal object oriented coding recommendations, but is required in order to give the SVVC user access to the rich VMM feature set available within each of the SVVC sub components.

Below is shown how the test bench can access the objects inside the SVVCs. In this case, the *dtst* transactor is used to apply a DUV configuration just after reset.

```
class cl_myenv extends vmm_env_syo;
  …

  task automatic cfg_dut();
    super.cfg_dut();
    …
    // Apply DUT configuration (just after HW rst)
    if1_svvc.dtst.master_write(adr, data, …);
    if1_svvc.dtst.master_getresp(response, …);
    …
  endtask

  …
endclass
```

Figure 13 – Controlling SVVC Transactor

As in a standard VMM test bench, test cases are simply class extensions of the top level test bench environments. In doing this operation, test case specific code is added to mainly the extension of the *start()* method, but other methods may also be extended, for instance *cfg_dut()*.

The example below displays a simple random test case, in which the *blueprint* [2] technique is used to constrain the randomly generated transactions flowing to the DUV. Note how easily the SVVC concept integrates into the test case – the test case writer knows by default that the atomic generator inside the SVVC will be named *agen*. Any other engineer familiar with the SVVC concept, who is trying to read the test case, therefore easily understands what the test case does without having to search for objects names and their declaration in a huge test bench.

```
// Inherit cl_mysvvc_data class to add constraint

class cl_mysvvc_data_smalladdr extends cl_mysvvc_data;
  constraint co_addr {
    addr < 'h100;
  }
endclass

// Extend the test bench environment into test case

class cl_mytestcase extends cl_myenv;

  task automatic start();
    super.start();
    begin
      cl_mysvvc_data_smalladdr bp;
      bp = new();
      if1_svvc.agen.randomized_obj = bp;
    end
    if1_svvc.agen.stop_after_n_insts = 10;
    if1_svvc.agen.start_xactor();
    …
  endtask

  …
endclass
```

Figure 14 – Test Case Accessing SVVC Resources

9.0 Industry Experience

On a large ASIC project, in the range of 25 SVVCs were developed to accommodate the need for block and system level verification. Experience shows that well trained SVVC developers are capable of writing a new on-chip protocol SVVC from scratch in less than 1-2 weeks, whereas engineers new to the SVVC concept needed in the range of 3-4 weeks.

With the verified SVVCs in hand, other engineers could easily design VMM test benches for the RTL blocks. Focus could be placed on the test bench specifics, such as reference models, scoreboards, functional coverage as well as assertions, while simply plugging in the SVVCs as previously described, without worrying about the exact implementation of the physical protocol. Once the block test bench was ready for a module, the RTL designer could perform initial simulation based bring-up of her/his block, largely without having to debug the test bench, as the SVVCs were already verified. In the order of a few weeks, quite complex test benches could be composed. Also, engineers with little or no prior knowledge of VMM found it easy to integrate simple SVVC based VMM test benches, and were capable of doing so in quite a short time.

On top of the SVVCs and the test benches, engineers with no experience or in-depth knowledge of these elements were able to create both simple directed test cases as well as constrained random test cases while only being exposed to a limited learning curve. This shows that a project or company wide decision of creating VIP and test benches based on the SVVC concept yields a

superior productivity and usability for the average engineers when compared to just using VMM out-of-the box.

10.0 Conclusion

The VMM framework is very flexible and allows verification components to be written in many different ways. However, it can be difficult to combine and reuse existing verification components coming from different sources and it often requires a considerable amount of effort to glue them together.

The concept of SystemVerilog Verification Components (SVVCs) provides a standard way of writing VIP and ensures that no glue is required. Components can be reused without modification from the block level to the system level in a scalable way. Restrictions on the VMM feature set provide uniformity and avoid some common pitfalls. With a little help from assertions an SVVC can verify itself, which enables test bench development to occur in parallel with the design.

Together, these features provide an efficient framework for the verification of large and complex designs, all the way from the block to the system level.

11.0 Acknowledgements

We would like to recognize the support we receive from Synopsys employees Martine Chegaray, Shawn Honess and Göran Larsson.

Thanks also go to Robert Fairlie (Verilab), Kasper Tonsberg (SyoSil) and Michael Andersen (SyoSil) for reviewing this paper and Lasse Lauridsen for valuable feed back in the early phases of conceiving the SVVC framework.

12.0 References

[1] *IEEE Standard for SystemVerilog*. IEEE 1800-2005.
[2] *Verification Methodology Manual for SystemVerilog*. Bergeron et al. Springer 2005.

About SyoSil:

SyoSil is a consulting company holding broad expertise within the field of System-on-Chip and ASIC solutions, including specification, methodologies, design and verification. We are specialized in verification strategies, advanced EDA verification tools including formal methods (property checking) and SystemVerilog for RTL design, assertions (SVA) and object oriented test benches (SVTB).