Off To The Races With Your Accelerated SystemVerilog Testbench

(A Methodology for Hardware-Assisted Acceleration of SystemVerilog Testbenches)

Hans van der Schoot, Anoop Saha, Ankit Garg, Krishnamurthy Suresh
Mentor Graphics Corporation
{hans_vanderschoot, anoop_saha, ankit_garg, k_suresh}@mentor.com

ABSTRACT

A methodology is presented for writing modern SystemVerilog testbenches that can be used not only for software simulation, but especially for hardware-assisted acceleration. The methodology is founded on a transaction-based co-emulation approach and enables truly single source, fully IEEE 1800 SystemVerilog compliant, transaction-level testbenches that work for both simulation and acceleration. Substantial run-time improvements are possible in acceleration mode and without sacrificing simulator verification capabilities and integrations including SystemVerilog coverage-driven, constrained-random and assertion-based techniques as well as prevalent verification methodologies like OVM or UVM.

General Terms

Verification, Performance.

Keywords

Acceleration, Emulation, Co-modeling, SystemVerilog, OVM/UVM.

1. INTRODUCTION

This paper describes a methodology for writing modern SystemVerilog testbenches that can be used not only for software simulation, but especially for hardware-assisted acceleration. Hardware-assisted speedup in testbench execution is compelling when one considers that ever growing verification complexity coupled with short time to market windows and scarce engineering resources make the need for fast simulation run times increasingly critical. For instance, think of viewing a full frame of graphics in a matter of minutes instead of a day of simulation. Simply put, faster testbenches enable longer and more test cases to be run in less time, allowing more requirements to be covered and more bugs uncovered.

Hardware-assisted testbench acceleration can in principle be achieved with full emulation through a fully synthesizable testbench, or more conventionally with co-simulation where an RTL DUT is mapped onto an emulation platform that interacts with the simulated testbench on a workstation at a clock cycle basis. With today's advanced transaction-level testbenches, however, the pragmatic approach is to have certain testbench components – the lower pin-level components like drivers, monitors etc. – synthesized into real hardware and running inside the emulator together with the DUT, while other non-synthesizable testbench components – the higher transaction-level components like generators, scoreboards, coverage collectors etc. – remain in software running inside the simulator. Communication between simulator and emulator is consequently transaction-based, not cycle-based, reducing communication overhead and increasing performance because data exchange is infrequent and information rich and high frequency pin activity is confined to run at full emulator clock rates.

The methodology presented herein promotes this so-called co-emulation (also known as co-modeling) approach and aims to maximize reuse between pure simulation-based verification and hardware-assisted acceleration. It enables truly single source, fully IEEE 1800 SystemVerilog compliant, transaction-level testbenches that work interchangeably for both simulation and acceleration. In acceleration mode it offers substantial run-time improvements while retaining all simulator verification capabilities and integrations. This includes in particular support for modern coverage-driven, constrained-random and assertion-based techniques in SystemVerilog as well as prevalent verification methodologies like OVM or UVM, and VMM. The subsequent sections lay out the details of and illustrate the proposed transaction-based acceleration methodology for SystemVerilog in terms of the testbench architecture and modeling rules and guidelines.

2. TERMINOLOGY

Co-emulation, or (transaction-level) co-modeling, is the process of modeling and simulating untimed behavioral models in conjunction with synthesizable hardware models running on an emulator, intercommunicating through transactions or function/task calls. The untimed transaction-based behavioral models are collectively referred to as the HVL side, while the cycle-accurate synthesizable hardware models constitute the HDL side.

SCE-MI 2, or Standard Co-Emulation Modeling Interface 2, is a set of standard modeling interfaces defined within Accellera for multi-channel communication between software models describing system behavior (i.e. the HVL side) and structural models describing the implementation of a hardware design (i.e. the HDL side). It is based on SystemVerilog-DPI as the foundation to realize communication between HDL code running in an emulator and C/C+/SystemC code running on a workstation.

A transactor is a component responsible for converting untimed transactions into series of cycle-accurate clocked events to be applied to a given pin interface, and/or conversely, for converting cycle-accurate pin activity observed into higher level transactions. In the specific context of hardware-assisted verification, a transactor is a SystemVerilog interface or module on the HDL side that has a signal-level interface with the DUT and a transaction-level interface with the HVL side. Transactors are sometimes also referred to as BFMs (Bus Functional Models) and the two terms are considered synonymous in this paper.

TBX™, or TestBench Xpress™, is the third generation hardware-assisted acceleration solution from Mentor Graphics, enabling state-of-the-art, comprehensive transaction-based co-emulation coupled to Mentor Graphics's Veloce emulation platform. It includes synthesis support of a rich extension of the RTL subset of SystemVerilog with behavioral clock generation and reset logic, initial and final blocks, implicit FSMs, SystemVerilog-DPI functions and tasks, synchronization events, waits, system tasks, and more, thereby offering maximum HDL modeling flexibility without performance penalties.

3. REQUIREMENTS

Several requirements are at play when devising a transaction-based acceleration methodology for SystemVerilog. Firstly, it must adhere to the principles of co-emulation which implies the need to partition a testbench into a synthesizable HDL side and a distinct HVL side handled by separate tools running on two different physical devices – emulator and workstation – and interacting at the transaction-level. The HDL side, then, must bear the limitations of modern day synthesis technology, and the communication with the HVL side must be fast and efficient so as to minimize impact on raw emulator performance.

Today's transaction-based testbenches like OVM/UVM testbenches have a layered foundation that exhibits a separation between timed and untimed (or partially timed) aspects of the testbench. As illustrated in Figure 1, a transactor layer forms the bridge between the cycle-accurate signal level of abstraction near the DUT and the transaction level of abstraction in the rest of the testbench. A co-emulation flow enforces this separation and requires that the transactor layer components are included on the HDL side to run alongside the DUT on the emulator. It further requires that the HDL and HVL sides are completely separated hierarchies with no cross module or signal references, and with the code on the HVL side strictly untimed. This means that the HVL side cannot include any explicit time advance statements like clock synchronizations, # delays and wait statements, which may occur only on the HDL side. Abstract event synchronizations and waits for abstract events are permitted on the untimed HVL side, and it is still time aware in the sense that the current time as communicated with every context switch from HDL to HVL side can be read. As a result of the HDL-HVL partitioning, performance can be maximized because testbench and communication overhead is reduced and all intensive pin wiggling takes place in the grey area in Figure 1 targeted to run at emulation speeds.

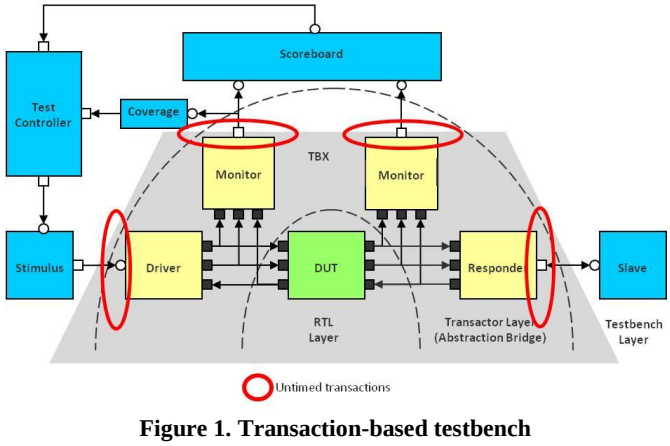

Figure 1. Transaction-based testbench

Another important methodology requirement is that it yields 'single-source' testbenches for both simulation and acceleration. This means that the HVL-HDL partitioning must function the same in co-emulation and in simulation alone, yet without the use of hooks like compile-time or run-time switches that would disable entire branches of code and pretty well implement two separate code bases. It also implies that the benefits of using SystemVerilog and verification methodologies like OVM or UVM for creating modular, reusable verification components and testbenches must be preserved along with associated simulator capabilities for analysis and debug. Key to achieving that proves to be the application of what is known in the object oriented world as a *remote proxy* design pattern. In this design pattern access to a remote object – e.g. a component on the HDL side – is controlled by a surrogate in the application domain – e.g. a component on the HVL side – through some indirect reference to uniquely access the remote object. Figure 2 illustrates this, where driver, responder and monitor components in blue act as proxies on the HVL side for the real transactors in yellow on the HDL side implementing synthesizable driver, responder and monitor BFMs, respectively. Communication between each transactor and its proxy occurs through a remote procedure invocation mechanism using BFM-like task and function calls, as detailed later. The mechanism is inspired by the known Accellera SCE-MI 2 function model and has the same kind of performance benefits as SCE-MI 2 [1]. This modeling practice in effect enables an acceleration methodology for SystemVerilog that is verification methodology neutral and thus applicable to OVM or UVM, and VMM.

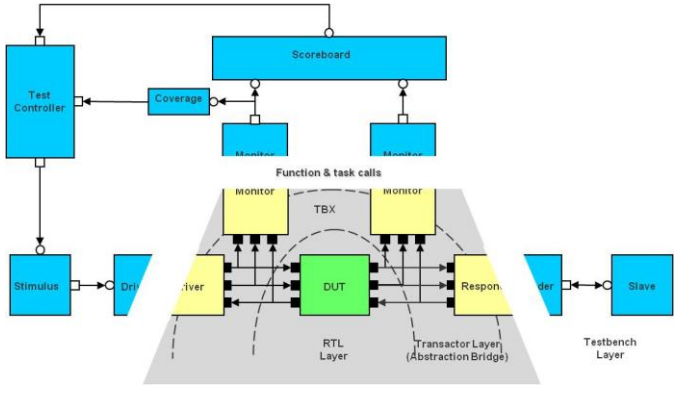

Figure 2. Transaction-based testbench with transactor proxies

A prior attempt towards enabling a methodology for accelerating SystemVerilog and OVM testbenches was made by Saha et al. in [5], proposing a considerably different use model for HVL-HDL communication referred to as XTLM (eXtended TLM). XTLM comprises a library of 'acceleration-friendly' TLM-based interface components in SystemVerilog, optionally derived from the TLM components in the OVM class library. It provides TLM fifos and channels, ports and exports that are enhanced for message passing across the HVL-HDL abstraction boundary using an intermediate C layer and SCE-MI 2 compliant SystemVerilog DPI-C. The rationale was that with the Accellera SCE-MI 2 standard already defining the communication semantics between HDL transactors and C models [1], XTLM implements an extra layer above the C layer to make the latter transparent to the user. Because of its usage of C as an intermediate language layer though, this approach naturally inherits the restrictions of that language.

In comparison, where XTLM enables a set of fabricated HVL-HDL connections built from the XTLM library components with a fixed API, the transaction transport mechanism presented in this paper utilizes exclusively built-in SystemVerilog constructs for a flexible user-defined API that is simpler and more intuitive and therefore generally easier to learn. And with the intermediate C layer gone, it

proposes just a small structural change at the boundary between DUT and testbench as part of the verification methodology used, where XTLM is structurally much more obtrusive. A detailed description of XTLM and usage examples can be found in [5].

4. THE METHODOLOGY

For a typical SystemVerilog testbench a single top level module encapsulates all elements of the testbench. This includes all verification environment components, clock and reset generators, the RTL DUT, and any SystemVerilog interfaces used to bundle the external pins of the DUT for access by environment components. In the common case of class-based verification components, such as OVM components, the access to the pins to drive or sample values is through a virtual interface handle – a pointer to a concrete interface. Virtual interfaces are the established means to connect an OVM testbench or any dynamic, object-oriented SystemVerilog testbench to a statically elaborated HDL model.

While this practice works fine for simulation it falls short for co-emulation, demanding two separated hierarchies – one synthesizable – that transact together without direct cross signal accesses. A methodology that does meet the requirements for co-emulation can be defined in terms of three high level steps as follows:

1. Employ two distinct HVL and HDL top level module hierarchies;
2. Identify the timed testbench portions and model for synthesis under the HDL top level hierarchy;
3. Implement a transaction-level interface between the HVL and HDL top level hierarchies.

The next sections describe each of these steps in detail.

4.1 Two Distinct Top Level Module Hierarchies

As the conventional single top testbench architecture is not suited for co-emulation, the first step is to rearrange and create dual HVL and HDL top level module hierarchies. This is conceptually quite simple, as shown in Figure 3. The HDL side must be synthesizable and

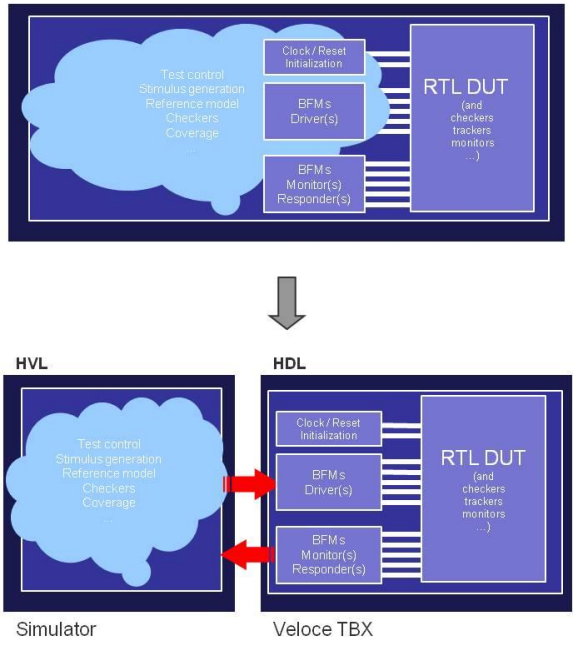

Figure 3. Separated HVL and HDL top level module hierarchies

should contain essentially all clock synchronous code, namely the RTL DUT, clock and reset generators, and the BFM code for driving and sampling DUT interface signals. The HVL side should contain all other (untimed) testbench code including the various transaction-level testbench generation and analysis components and proxies for the HDL transactors.

This modeling paradigm is facilitated by virtue of advancements made in synthesis technology across multiple tools. For example, Mentor Graphics' Veloce TBX™ provides technology that can synthesize not only SystemVerilog RTL but also implicit FSMs, initial and final blocks, named events and wait statements, import and export DPI-C functions and tasks, system tasks, memory arrays, behavioral clock and reset specification along with variable clock delays, assertions, and more. All supported constructs can be mapped on a hardware accelerator, and all models synthesized with Veloce TBX™ run at full emulator clock rate for high performance. Moreover, they can be simulated natively on any IEEE 1800 SystemVerilog compliant simulator. This synthesis advancement was a precursor to the SCE-MI 2 standard developed within Accellera to enable effective development of 'emulation-friendly' transactors [1].

Sample A.1 of the Appendix illustrates the rearrangement of a conventional single top hierarchy (module `top` in Sample A.1.a) into a dual HDL-HVL top hierarchy (modules `hdl_top` and `hvl_top` in Sample A.1.b) for co-emulation. This code example and subsequent code examples are based on a SystemVerilog testbench for a floating point unit (FPU) design adopted from the OVM cookbook [2]. As one can see, the FPU design and pin interface have moved to the HDL top level module (i.e. lines 10-17 and 12-19 in SampleA.1.a. and A.1.b), together with the clock generator (i.e. lines 26-33 and 21-25 in Sample A.1.a. and A.1.b). The clock generator has changed slightly with the use of a specific initial block in place of the non-synthesizable fork-join block.

A common package has also been introduced for convenient sharing of test parameters between the separate HDL and HVL top level hierarchies (i.e. lines 3-5 and 1-5, 10, 32 in Sample A.1.a. and A.1.b). The remainder of the single top hierarchy has been preserved under the HVL top level module including a virtual pin interface connection, now by hierarchical cross reference `hdl_top.fpu_if` into the HDL top level module (i.e. line 40 in Sample A.1.b). Certainly, neither a pin-level HVL-HDL interface nor an HVL-HDL cross module reference is permitted in the dual top co-emulation architecture, but this will be remedied in the next step where each transactor layer component is split into a synthesizable BFM on the HDL side and a corresponding untimed testbench component on the HVL side using a purely transaction-based communication mechanism.

It is worth pointing out that next to hardware-assisted acceleration there are other good reasons to adopt a dual top testbench architecture. For instance, it can facilitate the use of multi-processor platforms for simulation, the use of compile and run-time optimization techniques, or the application of good software engineering practices for the creation of highly portable, configurable VIP as discussed in [3].

4.2 Timed Testbench Modeled Under HDL Top

Forming the abstraction bridge between the timed signal level and untimed transaction level of abstraction, transactor layer testbench components like drivers, monitors or responders convert 'what is being transferred' into 'how it must be transferred', or vice versa, in accordance with a given interface protocol. The timed portion of

such a component is reminiscent of a conventional BFM, a collection of threads and associated tasks and functions for the (sole) purpose of translating to and from timed pin-level activity on the DUT. In SystemVerilog object-oriented testbenches this is commonly modeled inside classes, e.g. classes derived from the `ovm_driver` or `ovm_monitor` base classes in OVM. The DUT pins are bundled inside SystemVerilog interfaces and accessed directly from within these classes using the virtual interface construct. Virtual interfaces thus act as the link between the dynamic object-oriented testbench and the static SystemVerilog module hierarchy.

With regard to co-emulation, BFMs are naturally timed and must be part of the HDL top level module hierarchy, while dynamic class objects are generally not synthesizable and must be part of the HVL hierarchy. In addition, a transactor layer component usually has some high level code next to its BFM portion that is not synthesizable either, for example a transaction-level interface to upstream components in the testbench layer. All BFMs must therefore be 'surgically' extracted and modeled instead as synthesizable SystemVerilog HDL modules or interfaces.

Using this principle it is possible without much difficulty to write powerful state machines to implement synthesizable BFMs. Furthermore, when modeling these BFMs as SystemVerilog interfaces it is possible to continue to utilize virtual interfaces to bind the dynamic HVL and static HDL sides. The key difference with conventional SystemVerilog object-oriented testbenches is that the BFMs have moved from the HVL to the HDL side and the HVL-HDL connection must now be a transaction-level link between testbench objects and BFM interfaces. That is, testbench objects may no longer access signals in an interface directly, but only indirectly by calling (transaction-level) functions and tasks declared inside a BFM interface. This yields the testbench architecture already discussed briefly in Section 2 and depicted in Figure 2. It works natively in simulation and it has been demonstrated to work also in co-emulation (i.e. with Mentor Graphics' Veloce TBX™ acceleration solution). The next section details the concrete mechanism for HVL-HDL communication using remote function/task calls.

4.3 Transaction-Level HVL–HDL Interface

With the timed and untimed portions of a testbench fully partitioned, what remains is establishing a transaction-based communication mechanism for co-emulation. As suggested above, the use of virtual interface handles on the HVL side bound to concrete interface instances on the HDL side enables a flexible transaction transport mode for HVL-HDL communication provided thus that BFMs are implemented as SystemVerilog interfaces in the HDL hierarchy, not as modules. The flexibility stems from the fact that user-defined tasks and functions in these interfaces form the API.

Following the remote proxy design pattern discussed earlier, components on the HVL side acting as proxies to BFM interfaces can call relevant tasks and functions declared inside the BFMs via virtual interface handles to drive and sample DUT signals, initiate BFM threads, configure BFM parameters or retrieve BFM status. By retaining specifically the original transactor layer components like driver and monitor classes as the BFM proxies (see Figure 2) – minus the extracted BFMs themselves – impact on the original SystemVerilog object-oriented testbench is minimized. The proxies form a thin layer in place of the original transactor layer, which allows all other testbench layer components to remain intact. This offers maximum leverage of existing verification capabilities and methodologies.

The remote task/function call mechanism is based for the most part on the known Accellera SCE-MI 2 function model, and so it has the same kind of performance benefits as SCE-MI 2. In the traditional SCE-MI 2 function-based model it is the SystemVerilog DPI interface that is the natural boundary for partitioning workstation and emulator models [1], whereas the proposed methodology here uses the class object to interface instance boundary as the natural boundary for the same partitioning. Extensions specifically designed for SystemVerilog testbench modeling are added, most notably task calls in the workstation to emulator direction in which use of time-consuming/multi-cycle processing elements is allowed. This is essential to be able to model BFMs on the HDL side that are callable from the HVL side.

The HVL-HDL co-modeling interface mechanism is depicted in Figure 4. A proxy class `bus_driver` has a virtual interface handle `m_bfm` to a corresponding BFM model `bus_driver_bfm` implemented as a synthesizable interface. Time-consuming tasks and non-blocking functions in the interface can be called by the driver proxy via the virtual interface to execute bus cycles, set parameters or get status information. Notice the 'bfm' suffix in the BFM interface name, which is recommended as a naming convention. Also notice the use of the `bus` pin interface confined to the BFM by inclusion through its port list.

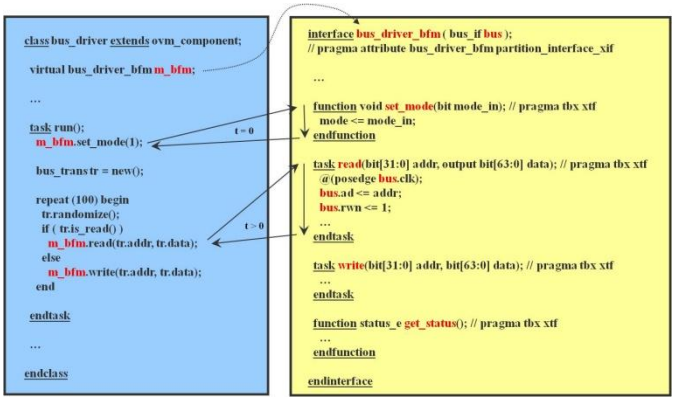

Figure 4. BFM interface with HVL proxy class

4.3.1 Transaction Object Conversion

Classes and other dynamic or unpacked data types in SystemVerilog are generally not synthesizable and can therefore not be used as BFM function/task arguments. For SystemVerilog object-oriented testbenches that extensively use class-based transactions (e.g. those derived from the `ovm_transaction` base class in OVM) it means that these transactions cannot simply be passed as is between the BFM interfaces and their proxies. However, since BFM functions and tasks are user-defined, it may be pertinent to pass transaction class members as individual packed arguments, just as shown in the code example of Figure 4 for the address and data attributes of bus transactions. Or one may choose to utilize special conversion routines to convert explicitly between class-based transactions and suitable packed type representations that are synthesizable such as a bit vector or packed struct. When utilized, it is recommended to standardize on `from_class(...)` and `to_class(...)` methods defined in an external converter class for each transaction type that must cross the HVL-HDL boundary. A concrete example is given in Sample A.2 of the Appendix for FPU request transactions.

Sample A.3 of the Appendix provides an example transformation of a purely class-based FPU monitor from the OVM cookbook example

kit [2] into a functionally equivalent BFM/proxy pair suited for both simulation and co-emulation. The FPU monitor proxy reimplements tasks `monitor_request()` and `monitor_response()` (i.e. lines 21-30 and 32-46 in Sample A.3.b) to call corresponding tasks in the BFM (i.e. lines 58-68 and 70-73 in Sample A.3.b) to perform the pin-level sampling of FPU request and response transactions and output these to the BFM proxy. External converter classes with `from_class(...)` and `to_class(...)` methods are used to convert between FPU transaction objects and convenient synthesizable packed struct representations of these transactions (i.e. lines 27 and 39 in Sample A.3.b), as shown in Sample A.2 for FPU requests.

For the example given it is assumed that the BFM interface is instantiated somewhere under the HDL top level hierarchy and that its corresponding proxy object on the HVL side has a virtual interface reference to the BFM. The actual binding of the virtual interface to the hierarchical HDL path of the BFM is not shown for brevity. Any such binding mechanism can be made to work also in the context of co-emulation. For OVM testbenches a recommended method described in [3] utilizes a general purpose OVM container class for wrapping any SystemVerilog type so that it can be used with the OVM configuration mechanism. It works just fine for binding BFM / proxy pairs.

4.3.2 HDL-to-HVL Back-pointers

For modeling flexibility and completeness a transaction-level HVL-HDL co-modeling interface can be defined in both directions. Similar to an HVL proxy class calling tasks and functions declared in an HDL interface, as discussed thus far, one can define how an HDL interface can call functions[1] declared in an HVL class. This would enable transaction-based HVL-HDL communication initiated from the HDL side. Specifically, a BFM interface may call relevant class member functions of its proxy object on the HVL side for instance to provide sampled transactions for analysis or indicate other status information. Figure 5 illustrates this. As shown, the handle of a BFM interface to the BFM proxy can be assigned simply inside the proxy itself via its virtual interface handle to the BFM. Access to any data members in the BFM proxy would not be permitted, just as cross signal references into the BFM are not allowed. Due to language restrictions on matching types, the proxy class definition together with any types it depends on must be imported inside the BFM interface via one or more packages.

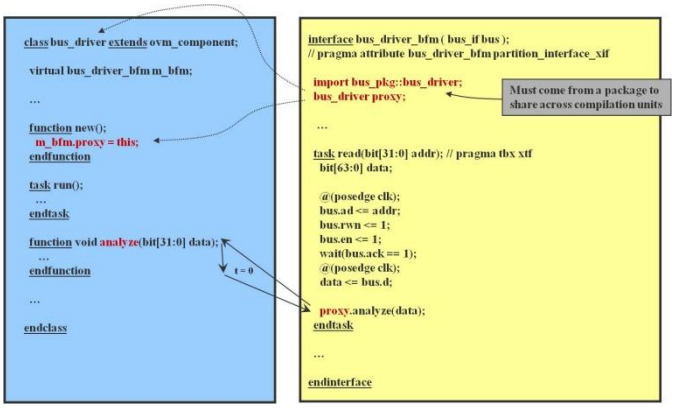

Figure 5. Driver BFM interface with HVL proxy

The use of such object handles in BFM interfaces back to their proxy classes, or 'back-pointers', is not firmly required for modeling reactive HVL-HDL communication. Yet this is particularly useful for components like monitors. A typical monitor continuously listens to an interface to extract transactions and pass them out to other testbench components for analysis, just like the FPU monitor in Sample A.3 of the Appendix. It initiates communication of observed transactions to 'subscribers' like scoreboards, coverage collectors or interrupt monitors. It is in effect more natural to have a monitor BFM 'push' instead of the BFM proxy 'pull' these transactions out. More importantly, doing so presents opportunities for significant performance optimization. Observed transactions are commonly distributed for analysis using void functions (e.g. the TLM `write(...)` function in OVM – i.e. line 46 in Sample A.3.a). Such one-way non-blocking calls can be dispatched and executed concurrently without even stopping the emulator clocks.

Sample A.4 of the Appendix provides a second take on remodeling the OVM-based FPU monitor for co-emulation. The monitor BFM now calls a void function `write` of its proxy via a back-pointer to push sampled FPU request-response pairs out to the HVL side (i.e. lines 62 and 22-26 in Sample A.4.b). The reader is invited to inspect the example in more detail with respect to the one in Sample A.3.

5. ADDITIONAL CONSIDERATIONS

Prying apart transactor layer components into synthesizable BFMs on the HDL side and untimed transaction-level proxy objects on the HVL side, as described in the previous section, has the consequence that the BFMs must be elaborated statically before run-time. At first sight some of the capabilities of a truly dynamic testbench may seem lost. Recall though that it is only the timed interface protocol that is to be implemented on the HDL side. Since the DUT interface and protocol are largely static there is no real loss of functionality. The idea is to retain the bits and pieces that must be dynamic inside the BFM proxy under the HVL top level module hierarchy. It should be apparent that a BFM interface is then in principle controllable completely through its dynamic proxy, via remote function or task calls. For instance, in terms of OVM it means that while BFMs cannot be created using the OVM factory or configured using the OVM configuration mechanism, the BFM proxies can be controlled in this way and hence indirectly the static BFMs themselves.

Thanks to the application of the remote proxy design pattern, prevalent testbench topology practices can also be facilitated without much alteration. Figure 6 depicts the normal view of an OVM agent for simulation and the adapted view for co-emulation. From the perspective of the OVM testbench on the HVL side there is no difference. Certainly, a matching topology of BFM interfaces under the HDL top can be configured only statically at elaboration-time, but as suggested by the code example in Figure 7 it is rather straightforward to employ SystemVerilog conditional or loop generate constructs on the HDL side in combination with a shared package of static test parameters imported and used by both HDL and HVL sides. The topology of a typical testbench is after all static in nature since it is expected to be fully elaborated before any testbench component starts running (e.g. the 'end-of-elaboration' phase in OVM executes before the 'run' phase). In case a truly dynamic alternative is desired it is possible to elaborate a fixed number of BFMs on the HDL side of which only a subset become active as maintained by the type and number of dynamically created BFM proxy objects.

[1] Since clocks are running exclusively on the HDL side, only functions – not tasks – should be called from HDL to HVL side. Strictly speaking only time-consuming tasks are problematic, but it is recommended to avoid tasks altogether.

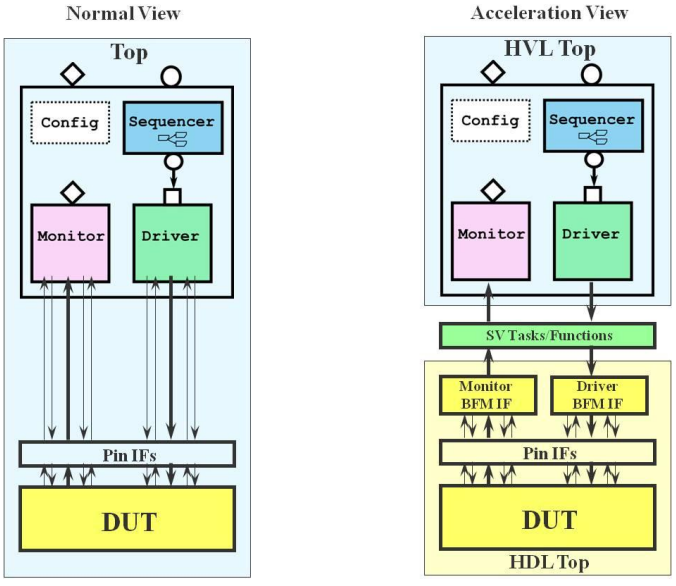

Figure 6. Simulation and co-emulation view of an OVM agent

Another methodology consideration is that current synthesis technology does not readily handle SystemVerilog coverage groups. Coverage groups are well suited for implementing transaction-level coverage concerned with the higher level functional requirements of a design. This stands in contrast to assertion coverage which lends itself for measuring the occurrence of lower level physical events involving the sampling of DUT signals and state variables, potentially over multiple consecutive clock cycles [4]. Assertion coverage fits naturally for BFMs and is in fact supported for synthesis by Veloce TBX™. Moreover, while surely coverage groups could be of use in BFMs as well, key to handling any genuine transaction-level coverage requirement for a BFM interfaceis once again the BFM's HVL proxy object, which may have coverage groups itself and forward transactions to other transaction-level coverage analysis components (e.g. see Sample A.4.b).

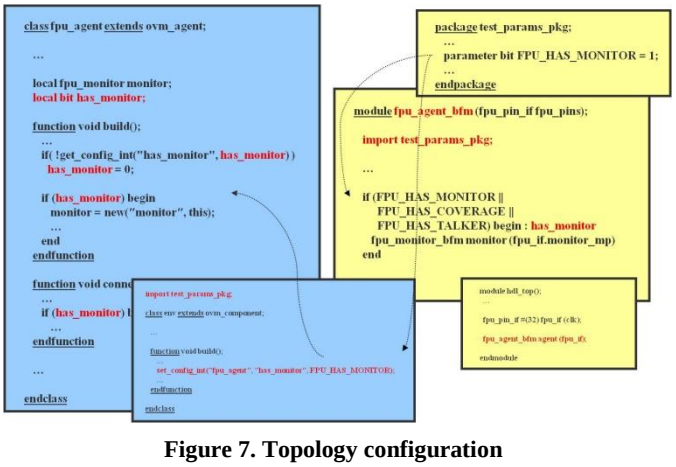

Figure 7. Topology configuration

6. EMPIRICAL RESULTS

Table 1 lists empirical results of applying the proposed transaction-based SystemVerilog testbench acceleration methodology. For several different designs the run-times for executing a test with pure simulation and with co-emulation are compared. The co-emulation engine used is Mentor Graphic's Veloce TBX™. The results clearly indicate that co-emulation can be much faster than simulation alone.

Table 1. Empirical results

Design	Simulation Time	Veloce TBX™	Speed-up Factor
Face Recognition Engine (1 MG)	½ hr.	6.58 secs.	128x
Wireless MM Sub-system (1 MG)	53 hrs.	658 secs.	288x
Menory Controller (1.1 MG)	5 hrs.	308 secs.	60x
Mobile Display Processor (1.2 MG)	5 hrs.	46 secs.	399x
Network Switch (34 MG)	16½ hrs.	240 secs.	245x
Graphics Sub-system (8 MG)	86½ hrs.	635 secs.	491x

Therefore, if simulation leaves you with insufficient throughput to meet your verification requirements, rather than taking calculated risks and limit the length of your simulation runs, you could greatly improve verification throughput with realistic tests using co-emulation.

7. SUMMARY AND CONCLUSIONS

A methodology was described for writing SystemVerilog and OVM or UVM testbenches that can be used not only for software simulation, but especially for hardware-assisted acceleration. For modern transaction-level testbenches, the pragmatic approach to hardware-assisted speedup in testbench execution is to have certain testbench components – the lower pin-level components like drivers, monitors etc. – synthesized into real hardware and running inside the emulator together with the DUT, while other non-synthesizable testbench components – the higher transaction-level components like generators, scoreboards, coverage collectors etc. – remain in software running inside the simulator. Communication between simulator and emulator is then transaction-based, not cycle-based, reducing communication overhead and increasing performance because hardware-software data exchange is infrequent and information rich, and high frequency pin activity is confined to run in hardware at full emulator clock rates.

This so-called co-emulation or co-modeling approach is at the core of the methodology presented, which further maximizes reuse between pure simulation-based verification and hardware-assisted acceleration through the application of an object-oriented remote proxy design pattern. As a result, truly 'single source' and fully IEEE 1800 SystemVerilog compliant transaction-level testbenches can be created to work interchangeably for both simulation and acceleration. In acceleration mode substantial run-time improvements are made possible and without sacrificing simulator verification capabilities and integrations such as modern coverage-driven, constrained-random and assertion-based techniques and tools. Additionally, the acceleration methodology is independent of the SystemVerilog verification methodology used and applicable to all prevalent methodologies today including OVM or UVM, and VMM.

In technical summary, the proposed simulation and acceleration methodology stipulates that a testbench be partitioned into two completely separated hierarchies, a synthesizable HDL side and a strictly untimed HVL side. Cross module and signal references are not permitted between the two sides. Instead, only transaction-level data exchange is performed via 'remote procedure invocation' in

SystemVerilog, and with Accellera SCE-MI 2 inspired performance benefits. Specifically, each DUT interface protocol – or BFM – on the HDL side is modeled as a synthesizable SystemVerilog interface with designated tasks and functions that can be called from the HVL side through a virtual interface by a dynamic class object that acts as HVL proxy for the BFM. Transaction objects may thereby need to be converted into synthesizable arguments. Conversely, the BFM interface may also have an object handle back to its proxy to call functions defined in the proxy. Reactive transaction-based communication is thus supported across the HVL-HDL boundary in both directions with either the HVL proxy or the BFM as call initiator. Each pair of BFM and proxy is to be viewed essentially as a joint pair representing a single transactor.

8. ACKNOWLEDGMENTS

Thanks to our colleagues John Stickley and Russell Vreeland for their careful review of this paper.

9. REFERENCES

[1] Accellera – Interfaces Technical Committee, "Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual," Version 2.1 (Review Copy), October 21, 2010

[2] M. Glasser, "Open Verification Methodology Cookbook," Springer, 2009. (Associated example kit available at www.ovmworld.org/contribution-detail/24891)

[3] A. Rose, M. Glasser, B. Osman, "OVM Configuration and Virtual Interfaces," White Paper, Mentor Graphics, 2010.

[4] H. van der Schoot, J. Bergeron, "Transaction-Level Functional Coverage in SystemVerilog," DVCon, 2006.

[5] A. Saha, K. Suresh, A. Jain, V. Kulshrestha, S. Gupta, "An Acceleratable OVM Methodology Based on SCE-MI 2," DVCon, 2008.

Appendix – Code Samples

Sample A.1. From conventional single top to dual top hierarchy for co-emulation

(a) Single top hierarchy

```
1     module top;
2
3       parameter int HALF_PERIOD = 12;
4       parameter int DATA_SIZE = 8;
5       parameter int ADDR_SIZE = 10;
6
7       env #(DATA_SIZE, ADDR_SIZE) e;
8       fpu_vif fpu_vif_obj;
9
10      bit clk;
11      fpu_pin_if #(32) fpu_if (clk);
12
13      fpu fpu_dut(fpu_if.clk,
14                  fpu_if.opa,
15                  ...,
16                  fpu_if.snan
17                 );
18
19      initial begin
20        e = new("env");
21
22        fpu_vif_obj = new(fpu_if);
23        set_config_object(
24          "*", "fpu_vif", fpu_vif_obj, 0);
25
26        // start the clock running
27        clk = 0;
28        fork
29          forever begin
30            #HALF_PERIOD;
31            clk = !clk;
32          end
33        join_none
34
35        run_test();
36      end
37
38    endmodule
```

(b) Dual top hierarchy

```
1     package test_params_pkg;
2       parameter int HALF_PERIOD = 12;
3       parameter int DATA_SIZE = 8;
4       parameter int ADDR_SIZE = 10;
5     endpackage
6
7
8     module hdl_top;
9
10      import test_params_pkg::*;
11
12      bit clk;
13      fpu_pin_if #(32) fpu_if (clk);
14
15      fpu fpu_dut(fpu_if.clk,
16                  fpu_if.opa,
17                  ...,
18                  fpu_if.snan
19                 );
20
21      // tbx clkgen
22      initial begin // Clock generator
23        clk = 0;
24        forever #(HALF_PERIOD) clk = ~clk;
25      end
26
27    endmodule
28
29
30    module hvl_top;
31
32      import test_params_pkg::*;
33
34      env #(DATA_SIZE, ADDR_SIZE) e;
35      fpu_vif fpu_vif_obj;
36
37      initial begin
38        e = new("env");
39
40        fpu_vif_obj = new(hdl_top.fpu_if);
41        set_config_object(
42          "*", "fpu_vif", fpu_vif_obj, 0);
43
44        run_test();
45      end
46
47    endmodule
```

Sample A.2. Converting transaction objects for co-emulation

```
class fpu_request extends ovm_transaction;

  shortreal a;
  shortreal b;
  rand op_t op;
  rand round_t round;

  ...

endclass


package fpu_trans_util_pkg;
  typedef struct packed {
    bit [31:0] a;
    bit [31:0] b;
    op_t op;
    round_t round;
  } fpu_request_s;

  typedef bit [$bits(fpu_request_s)-1:0]
    fpu_request_vector_t;

  ...

endpackage


class fpu_request_converter;

  function void to_class(
      output fpu_request req,
      input fpu_request_vector_t v);
    fpu_request_s s = v;
    req = new();
    req.a = $bitstoshortreal(s.a)
    req.b = $bitstoshortreal(s.b);
    req.op = s.op;
    req.round = s.round;
  endfunction

  function void from_class(
      input fpu_request req,
      output fpu_request_vector_t v);
    fpu_request_s s;
    s.a = $shortrealtobits(req.a);
    s.b = $shortrealtobits(req.b);
    s.op = req.op;
    s.round = req.round;
    v = s;
  endfunction

endclass
```

Sample A.3. Transforming an FPU monitor for co-emulation

(a) Original monitor

```
class fpu_monitor extends ovm_component;

  ovm_analysis_port #(fpu_pair) pair_ap;

  // VIF handle to pin interface
  local virtual fpu_pin_if #(32) m_fpu_pins;

  ...

  function void connect();
    ... // Retrieve m_fpu_pins vif handle
  endfunction

  task run();
    fork
      monitor_request();
      monitor_response();
    join
  endtask

  task monitor_request();
    forever begin
      fpu_request req = new();

      do
        @(posedge m_fpu_pins.clk);
      while (m_fpu_pins.start != 1);

      req.a = $bitstoshortreal(m_fpu_pins.op_a);
      req.b = $bitstoshortreal(m_fpu_pins.op_b);
      req.op = op_t'(m_fpu_pins.fpu_op);
      req.round = round_t'(m_fpu_pins.rmode);

      $cast(m_req_in_process, req.clone());
    end
  endtask: monitor_request

  task monitor_response();
    forever begin
      fpu_response rsp = new();
      fpu_pair pair;

      ... // Timed code to sample response

      pair = new(m_req_in_process, rsp);
      pair_ap.write(pair);
    end
  endtask: monitor_response

endclass
```

(b) XRTL monitor BFM with proxy

```
class fpu_monitor extends ovm_component;

  ovm_analysis_port #(fpu_pair) pair_ap;

  // VIF handle to XRTL BFM
  local virtual fpu_monitor_bfm m_bfm;

  ...

  function void connect();
    ... // Retrieve m_bfm vif handle
  endfunction

  task run();
    fork
      monitor_request();
      monitor_response();
    join
  endtask

  task monitor_request();
    forever begin
      fpu_request req;
      fpu_request_s req_s;

      m_bfm.monitor_request(req_s);
      req_converter.to_class(req, req_s);
      $cast(m_req_in_process, req.clone());
    end
  endtask: monitor_request

  task monitor_response();
    forever begin
      fpu_response rsp;
      fpu_response_s rsp_s;
      fpu_pair pair;

      m_bfm.monitor_response(rsp_s);
      rsp_converter.to_class(rsp, rsp_s);
      ...
      pair = new(m_req_in_process, rsp);
      pair_ap.write(pair);
    end
  endtask: monitor_response

endclass


interface fpu_monitor_bfm(fpu_pin_if fpu_pins);
// pragma attribute fpu_monitor_bfm
    partition_interface_xif

  ...

  wire clk = fpu_pins.clk;

  task monitor_request(output
      fpu_request_s req); // pragma tbx xtf
    @(posedge clk);
    while (fpu_pins.start != 1)
      @(posedge clk);
    req.a = fpu_pins.op_a;
    req.b = fpu_pins.op_b;
    req.op = op_t'(fpu_pins.fpu_op);
    req.round = round_t'(fpu_pins.rmode);
  endtask

  task monitor_response(output
    fpu_response_s rsp); // pragma tbx xtf
    ... // Timed code to sample response
  endtask

endinterface
```

Sample A.4. Transforming an FPU monitor for co-emulation (take 2)

(a) Original monitor

```
1   class fpu_monitor extends ovm_component;
2
3     ovm_analysis_port #(fpu_pair) pair_ap;
4
5     // VIF handle to pin interface
6     local virtual fpu_pin_if #(32) m_fpu_pins;
7
8     ...
9
10    function void build();
11      ... // Retrieve m_fpu_pins vif handle
12    endfunction
13
14    task run();
15      @(posedge m_fpu_pins.clk);
16      fork
17        monitor_request();
18        monitor_response();
19      join
20    endtask
21
22    task monitor_request();
23      forever begin
24        fpu_request req = new();
25
26        do
27          @(posedge m_fpu_pins.clk);
28        while (m_fpu_pins.start != 1);
29
30        req.a = $bitstoshortreal(m_fpu_pins.op_a);
31        req.b = $bitstoshortreal(m_fpu_pins.op_b);
32        req.op = op_t'(m_fpu_pins.fpu_op);
33        req.round = round_t'(m_fpu_pins.rmode);
34
35        $cast(m_req_in_process, req.clone());
36      end
37    endtask: monitor_request
38
39    task monitor_response();
40      forever begin
41        fpu_response rsp = new();
42        fpu_pair pair;
43
44        ... // Timed code to sample response
45
46        pair = new(m_req_in_process, rsp);
47        pair_ap.write(pair);
48      end
49    endtask: monitor_response
50
51  endclass
```

(b) XRTL monitor BFM with proxy

```
1   class fpu_monitor extends ovm_component;
2
3      ovm_analysis_port #(fpu_pair) pair_ap;
4
5     // VIF handle to XRTL BFM
6     local virtual fpu_monitor_bfm m_bfm;
7
8     ...
9
10    function void connect();
11  ..  // Retrieve m_bfm vif handle
12      m_bfm.proxy = this;
13    endfunction
14
15    task run();
16      fork
17        m_bfm.request_daemon();
18        m_bfm.response_daemon();
19      join
20    endtask
21
22    function void write(fpu_pair_s pair_s);
23      fpu_pair pair = new();
24      pair_converter.to_class(pair, pair_s);
25      pair_ap.write(pair);
26    endfunction
27
28  endclass
29
30
31  interface fpu_monitor_bfm(fpu_pin_if fpu_pins);
32    // pragma attribute fpu_monitor_bfm
        partition_interface_xif
33
34    ...
35
36    import fpu_tlm_pkg::fpu_monitor;
37    fpu_monitor proxy;
38    // pragma tbx oneway proxy.write
39
40    fpu_request_s req_in_process;
41
42    task request_daemon(); // pragma tbx xtf
43      ... // Sample requests (req_in_process);
44    endtask
45
46    task response_daemon(); // pragma tbx xtf
47      fpu_pair_s pair;
48
49      @(posedge clk);
50
51      forever begin
52        @(posedge clk);
53        while (fpu_pins.ready != 1)
54          @(posedge clk);
55
56        ...
57
58        pair.req = req_in_process;
59        pair.rsp.result = fpu_pins.outp;
60
61        ...
62
63        proxy.write(pair);
64      end
65    endtask
66
67  endinterface
```