Reusing UVM Testbenches in a Cycle Simulator

A Hybrid Test bench Co-Simulation Solution Explained

Kristina Hager
International Business Machines
Server & Technology Group
Austin, Texas, USA
hagerk@us.ibm.com

Andrew Lynch, Umer Yousafzai, Carter Alvord
Cadence Design Systems
Systems and Verification Group
San Jose, CA, USA
drew@cadence.com; umer@cadence.com;
alvord@cadence.com

***Abstract*— IBM requires a solution to enable the re-use of SystemVerilog testbenches which accompany imported Verilog design IP in their larger custom verification environment. These SystemVerilog testbenches need to be integrated with IBM's custom C++ testbenches and run in IBM's cycle simulator on the entire DUT. This paper will present a high level view of our co-simulation solution which simulates the SystemVerilog testbenches by a commercially available event simulator and simulates the C++ testbenches and DUT in IBM's cycle simulator. This paper will address challenges resolved in creating this solution including synchronizing the event driven test bench with the cycle simulator, synchronizing test bench phases, connecting SystemVerilog test bench signals to the design, communicating error messages between the simulators, handling tool errors gracefully, and coordinating end of test status and simulator exit. Additionally, this paper will summarize the architecture of the solution enabled by the versatility of the UVM, DPI, VPI and IBM methodologies, present future challenges, show performance measurements of the initial solution, and suggest how these techniques could be used in different SystemVerilog test bench reuse scenarios.**

Keywords—functional verification; co-simulation; cycle simulation; event simulation; Universal Verification Methodology

I. INTRODUCTION

A. *Introduction and motivation*

Microprocessor designs today consist of SOCs which integrate design IP from diverse sources like memory subsystems, PCI Express, and on-chip peripheral interfaces. For design verification, IBM designers use a cycle based simulation methodology enabled by custom simulation tools tuned to their design styles. The new challenge that the IBM verification teams face is the integration of the event driven SystemVerilog test bench code which accompanies IBM created IP blocks or off-the-shelf Verilog design into a full chip environment with their custom C++ test benches attached to their cycle simulation environment.

IBM solved this integration problem by co-simulating a UVM test bench running in an event driven SystemVerilog simulator with their cycle simulator. This approach has significant advantages: IBM engineers can avoid re-implementing existing SystemVerilog test benches. IBM can also leverage commercial SystemVerilog simulators instead of developing their own SystemVerilog constraints solvers and simulation engine. We call this co-simulation solution the Hybrid Testbench or HTB.

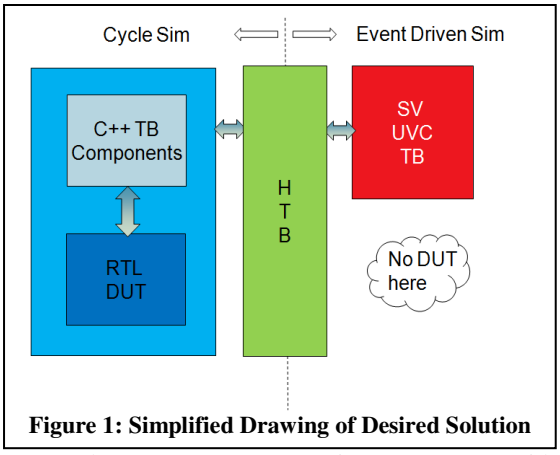


Figure 1: Simplified Drawing of Desired Solution

The solution described here connects a SystemVerilog engine to a cycle simulator. This solution can be generalized to other engines simulating a DUT with a C or C++ interface. We will enumerate and address the challenges we encountered in creating this solution. We took advantage of the versatility enabled by the DPI, VPI, UVM and IBM technologies to provide this solution in a handful of self contained packages. We will elaborate on these techniques and show how this simplifies the user view of the solution. We will also provide the results of performance experiments on the solution as provided to an IBM internal design team.

In this paper, we will often refer to the IBM cycle simulator as the "Mesa simulator" which is the internal name for their cycle simulation tool and the cycle simulation environment in general. IBM has developed a C++ based verification environment and library called "Fusion" which is similar in man respects to UVM. Our co-simulation solution was created with Cadence's event simulator known as "ncsim" or "IUS", so

we will use those terms interchangeably here to indicate the "event simulator" as well. Again, we will use the term Hybrid Testbench or the acronym HTB to refer to the co-simulation solution.

This paper should be of interest to designers and verification engineers seeking to reuse SystemVerilog test bench components with different design abstractions (untimed C/C++ models) or connecting SystemVerilog testbenches to algorithmic/modeling engines such as Matlab.

B. *Related Work*

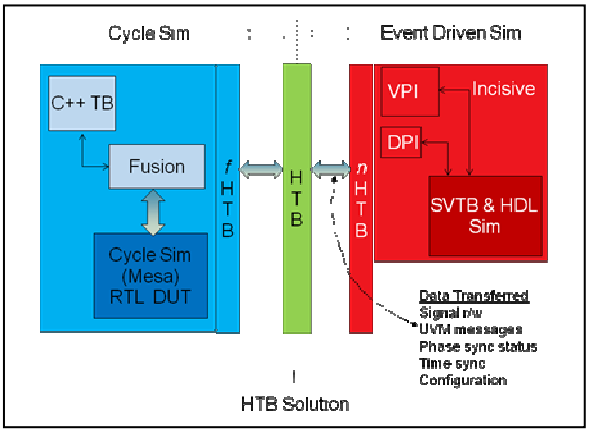
IBM has significant experience in developing co-simulation solutions. Experiences in these other approaches have helped inform the solution discussed here. We will briefly compare and contrast these co-simulations with the one discussed here.

One existing co-simulation solution, FusionNC, has the entire DUT and some SystemVerilog test benches simulated in an event driven simulator with the C++ test bench components handled by IBM's cycle simulation based test bench environment. The FusionNC solution could be considered the "opposite" of the solution presented here (where the DUT is entirely in IBM's cycle simulation environment). FusionNC is used when testbench components already exist in C++ or where it's easier to create them there. In this case, the DUT is simulated in ncsim either because the netlist is encrypted and can't be recompiled for the cycle simulator or when the design lends itself better to event simulation.

Another existing co-simulation solution, Cycle Cosim, utilizes two simulators each containing both test bench and design components: The event driven simulation has test bench and design components where it is assumed that the SystemVerilog test bench primarily accesses the Verilog design. The cycle driven simulation also has both C++ test bench components and its own, separate design components where it is assumed the C++ test bench primarily accesses the design in the cycle simulator. The two design pieces under simulation connect to each other at the IO level via a "bridge". The verification engineer must carry out steps to determine which signals need to be connected between the two designs and create this bridge. If there is any need for the SystemVerilog test bench to access the design running in the cycle simulator, then those signals must be manually brought out to the IO level "bridge" between the two models. In the solution presented in this paper, there is no need to manually create a "bridge" as there is only one unified DUT. While the SystemVerilog testbench will normally only access signals in its corresponding Verilog design IP, it can also formulaically access any signal in the design either inside or outside of the corresponding design IP.

Cadence has developed many co-simulation solutions to simulate designs in different engines such as analog solvers or a hardware emulator while other portions of the design or testbench simulates in an event driven digital software simulator. These experiences enabled the team to collaborate with IBM to propose a practical solution that builds on standard interfaces to enable an exemplary co-simulation approach.

Fig. 2 Simplified Drawing of Solution Architecture.



II. SOLUTION ARCHITECTURE

A. *Approach*

Before we began to create the HTB solution, the IBM and Cadence technical teams conducted brainstorming sessions regarding the breadth of usage scenarios of our current simulation tools, which of these usage scenarios the HTB simulation might need to support, and the best software architecture for the planned co-simulation solution. We considered multithreading approaches, multiple process approaches and various means of communicating information between the simulators such as pipe or shared memory approaches. As an example of a key usage scenario that drove an architectural decision, both IBM's and Cadence's simulation environments support features such as checkpoint and restart. However, both support this methodology in notably different ways. The end result of this discussion is that architecting the HTB simulation as a multi-process rather than a multi-threaded solution would retain the greatest flexibility. Similarly, the multi-process approach reduces complexity and improves ability to debug the simulations since each simulator will have its own process and memory space.

HTB actually uses three processes. The HTB simulation starts with a single process which sets up the HTB Core Library infrastructure and then initiates the two simulator processes. After the simulation processes are dispatched, the parent process simply waits for the two simulator processes to return. During the waiting period, the parent process can also monitor the children simulator processes. When both child processes return, the parent process executes some clean up code to ensure that shared resources are released.

The HTB software architecture can be decomposed into a three layer solution which we describe in this section. This separation of the solution into five pieces reduces complexity from a component dependence point of view and enables re-use from a simulator point of view. Fortunately, these components are well hidden from the end user.

B. HTB-Core Library

The HTB Core Library provides a shared object which implements a C API with APIs covering the requirements to: pass and retrieve messages, pass and retrieve signal read or write requests, signal how much time should be advanced, or stop at sync points where either simulator can communicate status or optionally communicate that it has encountered a scenario where it needs to abort so the other simulator can handle this gracefully. This C API provides a unified interface to several internal classes which implement the needed functionality. The advantage of providing a C API interface via a shared Dynamically Loaded Module (DLM) is that any C or C++ based program can load this DLM and access the APIs. Any such program will also need to link against the header file(s) accompanying the DLM

1) *Posix Shared Memory and Semaphores*

The HTBlib implements the required functionality via Posix shared memory and semaphores. The advantage of Posix shared memory is that it is extremely high performance, e.g. it is as fast as memory local to the process since it is referenced in the same manner

2) *Sync Points*

We will explicitly introduce one technique used in our solution as it is referenced many subsequent explanations. First, each simulator in the HTB execution stops at a set of predefined, named points called “sync points”. At every “sync point” each simulator waits for the other simulator to reach that point before proceeding. At every sync point, each simulator can also communicate if it is ready for the next phase of simulation (as in UVM phase) or not and whether or not it would like to abort.

These sync points include one right after elaboration and before sim time begins and another at the end of simulation before cleanup. The simulators also stop at a few sync points during each "simulation interval" to retrieve or send information about the number of cycles to proceed during this interval, to exchange signal read/write updates and exchange messages, and to communicate status regarding phase. At any sync point, a simulator can communicate the need to abort the sim.

C. *Fusion-HTB Interface*

The Fusion-HTB interface is the layer which seamlessly integrates into the Fusion phasing and signal handling methodology in a manner very similar to how IBM's C++ based verification IP integrates into the Fusion methodology. This layer invokes the API's provided by the HTB Core Library to communicate status, retrieve signal read/write requests or UVM messages that ultimately originate from the Ncsim-HTB Interface, and so on.

D. Ncsim-HTB Interface

1) *Phase-Stage Synchronization*

The UVM methodology handles phasing in an object oriented way through a uvm_phase class. This class defines phase behavior, state, context and exposes interfaces/apis for users to use and extend phasing behavior. In addition, the uvm_component class defines virtual tasks and functions corresponding to each phase where a VIP can do work in a synchronized manner to other VIPs in a testbench as shown in Figure 3.

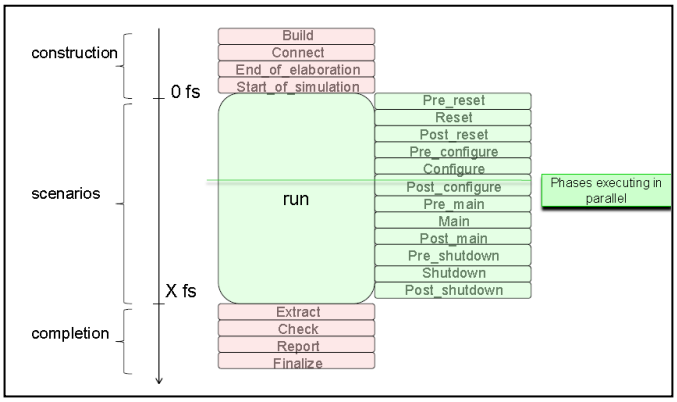


Fig. 3. UVM Phases

We take advantage of the uvm_objection feature along with uvm_phase apis to synchronize SystemVerilog testbench activity with stages in fusion. We add a class derived from uvm_test in the HTB package (called sync_phases) that gates the mapped phases from progressing using post_phase virtual tasks until fusion is also done with that stage. An instance of this class is required in the top-level Verilog wrapper or on the command line to provide the required synchronization with fusion stages .

```
task post_configure_phase(uvm_phase phase);
      phase.raise_objection(this);
      glob_uvm_phase_end = CONFIGURE;
      wait (glob_fusion_stage_sync == INIT_LOOP_DONE);
      phase.drop_objection(this);
endtask
```

Fig. 4. Mechanism to synchronize Phases and Stages

Using a post_phase method such as post_configure_phase allows us to utilize the property that *ALL* UVM testbench components are done with their configuration phase and are now in “transition” to the next phase. As shown in Figure 4, UVM testbench will be suspended until the glob_fusion_stage_sync signal has the value encoded by INIT_LOOP_DONE indicating that Fusion is ready to move to the next stage. The phase.drop_objection api allows UVM to proceed to the next time-consuming phase.

2) *Message Passing*

As part of inter-simulator testbench communication where the IBM simulator serves as the master, it is necessary to send UVM messages from the ncsim simulator to the IBM simulator for further processing. In order to do so, the HTB utilizes a custom report server derived from the UVM built-in *uvm_report_server*. The custom server is assisted by DPI-C code which is used to queue messages that need to be processed by the HTBlib layer.

Messages in UVM have an associated severity level and action. There are four severity levels including: information, warning, error, and fatal. Message actions can include sending to the display/log, counting message types, terminating the

simulation, or no specified action. Each severity level has an associated set of one or more default actions.

There is not a one-to-one match of message severity levels and actions between UVM and the IBM simulator. In UVM, error messages do not by default terminate the simulation. The user can override this behavior and require that a simulation terminate after a specified number of errors. In UVM, fatal messages always cause an immediate termination. The IBM simulator on the other hand does not distinguish between error and fatal conditions, defining only an error state. An error in the IBM simulator always leads to termination of the simulation.

A custom UVM report server is required to intercept UVM messages and their programmed actions and remap them to the corresponding IBM simulator severity levels. The following table provides the mapping between the two:

UVM Severity Level	IBM Simulator Severity Level
UVM_INFO	No mapping; not forwarded
UVM_WARNING	Warning
UVM_ERROR (non-terminating)	Warning
UVM_ERROR (terminating)	Error
UVM_FATAL	Error

The modifications made in the custom report server derived from the base uvm_report_server are minimal and well documented. If desired, users are able to create their own custom report servers as well, but they must inherit from the HTB custom report server and follow clearly defined instructions in order to preserve the above necessary functionality.

3) Signal Registration

In the HTB usage scenario, the Verilog design unit under test is connected to a larger design at the Verilog design's interfaces (inputs, outputs, and inouts). The design is simulated in the Mesa or IBM simulator. Therefore, communication between a testbench interface signal and a DUT port must be handled via the HTB interface. In an HTB simulation, the top level Verilog design module's portlist is retained, but the implementation of the design is removed. This port list represents the signals to be connected to the Mesa DUT. With a few exceptions, the module definition contains only a single initial block, which contains a call to the system task $htb_register_portlist. This system task does nothing when called during the flow of simulation, but its related compiletf, executed before simulation begins, performs consistency checks and configures the interface between ncsim and Mesa.

```
module dut(input wire [31:0] in1, output reg [1:0] out1);
   initial
     $htb_register_portlist();
endmodule // foo
```

The compiletf routine for $htb_register_portlist iterates over the port list of the containing module and builds a list of signals to be connected to Mesa. The direction of signal flow (Mesa to IUS, IUS to Mesa, or both) is taken from the port direction on the Verilog module. The compiletf routine checks that ports are of appropriate type. HTB currently only supports simple scalar and vector nets and registers. The routine also checks that the timescale requested by Mesa can be supported by the time precision of the Verilog portion of the design. All errors occurring during this phase are collected and reported before the simulation is terminated.

```
module foo;
   reg [31:0] a1, a2;
   MesaDut.a1 = a1;
   a2 = MesaDut.a2;
endmodule // foo

becomes

module foo;
   reg [31:0] a1, a2;
   $htb_register_write("MesaDut.a1",a1);
   $htb_register_read("MesaDut.a2",a2);
endmodule // foo
```

We also found that there were occurrences of "Out of Module References" or OOMRs which referred directly into the DUT from the SystemVerilog. For these we created $htb_register_read and $htb_register_write system tasks. These work much like $htb_register_portlist in that their compiletf routines add to the list of signals which are registered with the interface. The arguments consist of a string that represents the DUT path to the object of interest, and the object on the IUS side. We also found that macros are frequently used in referencing OOMR signals in the DUT, so we added $htb_define_macro("MacroName","MacroValue"); If the name string in $htb_register_read or $htb_register_write start with a ` character, we expand the following string with the defined macro. This eases the conversion of the original source code.

The VPI cannot directly drive nets. It can deposit values onto nets, but they will be overwritten as soon as any of the net's HDL drivers change value. Since inout ports are most often nets, we needed a means by which the VPI could drive the inout port. We did this by having the user add a register and a continuous assignment statement to the dummy DUT. The register is the same size and has the name of the inout port with "_data" appended. When the interface needs to read the port, it reads from the net. When it needs to drive the port, it writes to the register, and the continuous assign propagates the value to the net. $htb_register_read can find these signals automatically due to the defined name mapping.

```
module dut(input wire [31:0]in1, output
reg [1:0]out1, inout wire [31:0]biport);
   reg [31:0] biport_data;
   assign biport  = biport_data;
   initial
     $htb_register_portlist();
endmodule // foo
```

If requested, we will extend this to work with output wire ports as well.

We found that using the SystemVerilog port list as our interface definition can be limiting. Sometimes there are signals in that port list that require special handling, or signals which we do not want to attach to Mesa. We may in the future provide a mechanism (much like $htb_register_read and $htb_register_write) through which the user can individually register his signals, instead of using $htb_register_portlist. These tasks could take additional arguments to further refine how individual signals are handled.

It is important that we know all the signals of interest ($htb_register_portlist, $htb_register_read, $htb_register_write) and their sizes before we initialize communication with Mesa. We make use of the order of simulation related callbacks to configure the interface between IUS and Mesa

1. compiletf routines: Collect port and argument lists to build the list of signals to communicate with Mesa. Each instance of a system task will cause a call to the compiletf routine for that system task, allowing us to process all of the related argument and port lists. All compiletf routines will be complete before the VPI End of Compile callback occurs.
2. VPI End of Compile: Register the collected port and argument lists with Mesa.
3. VPI Start of Simulation: At this time we perform final configuration and set up the time related callback to pass control to Mesa
4. VPI After Delay: We use this callback to pass control to Mesa after a certain period of time.

When Mesa passes control to IUS, Mesa has the opportunity to specify a number of cycles for IUS to run. This number of cycles is converted into a delay using the cycle length and local time scale, and a VPI cbAfterDelay callback is scheduled. IUS simulates until this callback matures. The callback occurs, and control is passed back to Mesa.

Each time that control passes from Mesa to IUS, Mesa reads the output and inout signals from the DUT and passes the updates to the HTB component through a provided api. IUS reads this information from HTB, and drives it into the Verilog testbench. Similarly, each time that control passes from IUS to Mesa, IUS reads the input and inout signals from the testbench and passes the updates to the HTB component. Mesa then reads the data from the HTB component and drives it into the DUT.

We chose to use a simple one character per bit representation for both scalars and vectors. This was easier to code and debug, but does leave room for future optimization. At this point, all signals are read and written each cycle. This will be optimized such that only the signals that change will be copied.

III. Issues Considered

We discussed and resolved many issues regarding coordination and communication between the simulators. We present here these issues and our solution. These questions and answers can also serve as a FAQ.

1) How will time and cycles be advanced?

The IBM simulator is a cycle based simulator, and ncsim is an event based simulator which operates based on physical elapsed time. Therefore, the verification engineer must determine which clock in the cycle simulation is the one with the smallest clock cycle and also know the length of that cycle. The verification engineer specifies that time length to the ncsim simulator as the time elapsed during "one clock cycle length". The IBM simulator is the "master" in the HTB simulation and determines how many clock cycles to advance at each interval. That number of cycles is communicated to the ncsim simulator which translates the number of cycles into a length of time. The ncsim simulator runs for that length of time and then pauses to "sync" up with the IBM simulator.

2) How will the SystemVerilog simulator simulate an existing VIP/UVC with minimal changes and drive/sample the DUT in the cycle simulator?

The motivating usage scenario for HTB is the integration of an off the shelf Verilog design unit into a larger environment. This unit will be connected into a larger design at the Verilog design's interfaces (inputs and outputs and inouts). In the usual ncsim simulation the Verilog design and SystemVerilog testbench are compiled together such that the DUT ports are directly connected to the UVC's interfaces. However, in the HTB solution the DUT exists only in the IBM simulator so this connection no longer exists.

The verification engineer must make a minor modification to the top Verilog design component definition to exclude the definition of the design and include an invocation of a system task $htb_register_portlist which tells the ncsim-HTB component to read the portlist of the design. HTB samples values on input ports from the TB and drives the DUT input ports at the sync-points described in II A. It does the opposite for DUT output ports on the dummy DUT.

This question is answered more completely and with examples in section II.D.3 "Signal Registration".

3) How will the simulators stay in sync with respect to important execution points such as initialization, execution, and reporting of results?

The simulators have these important execution points pre-defined which are explained in the "sync points" section above. These sync points force each simulator to stop and wait at

those pre-defined points so that information can be exchanged between the simulators.

The simulators also utilize the few sync points enabled within each simulation interval. IBM VIP in C++ has its own concept of phasing which are called “Stages”. Once we mapped and understood the intended function of each phase and stage, we included code to keep the simulators in sync with respect to these phases and stages. For example, we didn’t want to start transaction collection and checking in the UVC checker until we knew that the design had come out of reset and that the config registers were programmed. During each interval, the ncsim simulator communicates via sync point to the IBM simulator whether it is "finished" or "busy" in that particular phase. If the IBM simulator receives a "busy" message from ncsim regarding a particular phase, then both simulators will stay in that phase/stage until all pieces of the test bench, including ncsim, report "finished". If the IBM simulator receives a "finished" from the ncsim simulator regarding a phase, and if all other test bench components are finished with that phase, then the IBM simulator will move to the next phase and signal to ncsim to do the same.

4) How will Incisive communicate UVM_ERROR type messages to the Mesa simulator?

The ncsim simulator can place UVM_ERROR type messages on a message queue maintained by the HTB core library at any time. At a pre-defined sync point, the IBM simulator will fetch messages from the queue. When the IBM simulator fetches messages from the queue, it will also examine the type of each message. At this time, the IBM simulator is configured to consider UVM_FATAL and `UVM_ERROR to be a sim terminating error. However, that configuration can be easily modified on either side.

5) How will each simulator communicate tool errors or the need to abort the sim?

Each "sync point" allows each simulator to send the name of the sync point it is waiting at, whether it is finished or busy in that phase, and a flag signaling if it needs to abort or not. If the simulator needs to abort immediately (immediately after the sync point) then it will set the abort flag. The other simulator checks for that flag and will also exit as gracefully as possible.

IV. HYBRID TESTBENCH CHALLENGES AND NEXT STEPS

A. Restrictions Placed on Solution

This approach is designed to minimize code changes in SystemVerilog testbench if the testbench complies with UVM guidelines. For instance, the SystemVerilog Interface is used to abstract connection between design and testbench. The SystemVerilog interface is also used to encapsulate properties and assertions. If the SV testbench is not UVM compliant and uses OOMRs for signal access/assertions, the testbench porting to HTB will require significant rewrite.

Secondly, this approach relies on a synchronous design which allows a clean mapping of a cycle in Mesa into a time quantum in event driven simulation.

B. UVM All the way but reality strikes

We encountered a style of OOMRs in SV VIP on a real project that we had to deal with. There was not any other better approach to make it easier to port to hybrid testbench. This is the case where a SV checker is monitoring something deep in the DUT hierarchy.

```
interface mytb_if;
    logic [31:0] data;
endinterface

module top;
  mytb_if tb_if();
  dut dut_i( .data (tb_if.data));
endmodule
```

Figure 5. Simple example of a need for hierarchical access

For example, Figure 5 shows a simple code fragment that illustrates the problem. *dut_i.data* in the simulator needs to be wired to *tb_if.data* for the SV checker to work. However, if the dut instance is embedded in the larger system, the port map doesn’t work and an OOMR access is required. To solve this problem we created a handful of system tasks to declaratively register an implicit read from or a write to a hierarchical path in the design from a corresponding signal in the interface.

Here is an example of registering an OOMR input signal:

```
$htb_define_macro("`MCP_PATH",
  "KMPU_MPU_UNIT_I.DDR34MC.DDR34LMC_CS_0.DDR34LMC");
$htb_register_read("`MCP_PATH.I_MC_CLOCK", mcp_sys_if.mcp_clk);
```

C. Pico seconds or a cycle?

In order for UVM testbenches to work properly in HTB, the verification engineer must properly pass to ncsim via a provided command line option how long a one clock cycle is in Mesa. It is dangerous to use # delays in SV TB to drivers or procedural code. For example, if the driver code in SystemVerilog has a # delay to model asynchronous behavior that is not mapped to a fast clock in the cycle simulator, the simulation will in all likelihood simulate incorrectly. There is nothing in HTB that detects this problem explicitly aside from checkers in the TB misfiring.

D. Future Challenges

The next phase of this project will require more complex data structure communication between event driven simulator and C++ testbench. For example, end-to-end checkers such as scoreboards are implemented in SV or in C++ and can reuse significant VIP logic by passing transactions across the language boundary using a TLM analysis port or export and auto transaction mapping from SV to C++ or vice-versa.

Corner case test scenarios also need better coordination of stimulus between different VIPs. For example, creating back-pressure on a FIFO by slowing the read frequency compared to the write frequency or creating system level congestion to a

shared resource such as DMA or memory controllers. Finer grain execution control and data sharing will be required to achieve this level of testing.

V. COST OF HTB SOLUTION

Once a verification team has understood the co-simulation approach described here at least at a high level, their next inquiry is always "How much extra time does this co-simulation take compared to the usual single simulation?". Clearly, the HTB co-simulation will necessarily cost more execution time and memory than the usual one simulator process. However, we need to be able to quantify that cost for the verification engineer end user.

We propose the following experiments to address this question. Since the initial usage scenario is a large IBM verification environment with a smaller SystemVerilog test bench, we will start our experiments with this in mind as described below:

- Mesa standalone: Capture baseline performance of a single Mesa (IBM) simulation without any HTB or SystemVerilog components. This is our starting point for analysis.
- Minimal HTB: Capture the performance of the same Mesa (IBM) simulation with the HTB functionality and ncsim process added but without any SystemVerilog testbench enabled. This experiment will show the cost of adding the HTB infrastructure and sync points every cycle alone. The ncsim process will still also be running, but it will not require any time for SystemVerilog test bench activities. In this mode, signal data will not be communicated between the simulation environments.
- Regular HTB: Capture the performance of the above experiment with the SystemVerilog test bench enabled. The ncsim process will need to elaborate and simulate the testbench. Comparing the performance of this experiment to the previous will show how much of the additional cost of an HTB sim is related to the SystemVerilog simulation and communication of things like signal reads and writes and message passing.

The goal of the above experiments is to quantify the minimum expected tax of an HTB simulation to execution time and memory and also to show an example benchmark using an existing use case. This data should also inform us (as the HTB developers) as to which parts of the HTB interface may need optimizations or, perhaps, where further improvements are not possible.

At this time, we have some preliminary results from an execution of the above experiment on a single HTB simulation environment. These experiments were run on a single Linux machine, so we do not need to account for system specification variations. However, these experiments were run on systems exposed to the fluctuations of a batch environment, so system load could have varied during the experiments. Given that only two sets of experiments were run on one simulation environment and the test machine was not otherwise idle, these results are not scientific. However, they are still a useful data point for both users and developers of HTB.

We found that the simulation time increased only slightly when moving from the “Mesa standalone” environment to the “Minimal HTB” environment. The increase in simulation time could be considered “in the noise” when running simulations on a batch system. However, we conclude that there indeed must be a small increase in CPU time when adding HTB and the ncsim process to the simulation environment. This is expected since the Mesa process will stop a few times each cycle to communicate with HTB. Also, the memory footprint of the “Minimal HTB” simulation grew by a small amount for the Mesa process. This is due to the shared memory segments for HTB being allocated and accounted to the Mesa process.

Next, we would like to compare the “Mesa standalone” simulation to the “Regular HTB” simulation. In this experiment, we used a cached elaboration on the ncsim side for best performance. In the “Regular HTB” simulation, the mesa process memory utilization grew by the same amount as for the “Minimal HTB” simulation. The wall clock time of the “Regular HTB” simulation increased to about 1.7x to 2.1x the wall clock time of the “Mesa standalone” process. We consider this to be a useful data point, but we also expect this factor to be largely dependent on the complexity and size of the SystemVerilog test bench, Fusion test bench, and the number of signals that are synchronized across the HTB interface. In other words, your mileage may vary..

VI. Summary

We have successfully built a co-simulation solution leveraging the standardization of the UVM methodology and base-class library, the flexibility of the VPI interface and the DPI interfaces in the Incisive simulator. In order to develop and debug this solution, IBM and Cadence decided to use a simple UVM example called “ubus” that contained all the necessary ingredients of a real life UVM environment. The example proved instrumental to prototype different ideas quickly and freely exchange code without worrying about IP security.

The HTB co-simulation solution is presently being used on an IBM SoC design where 100% of the active testbench is in IBM’s simulator and a complex checker from an IP-level SystemVerilog testbench is being re-used and simulated by ncsim to aid in debug and system-level checking.

ACKNOWLEDGMENT

We would like to acknowledge the following coworkers for their assistance in the development of the HTB solution: Ron Cash, Wolfgang Roesner, Walt Kowalski from IBM and Amit Kohli from Cadence.

References

[1] "The UVM User's Guide", UVMv1.1, uvmworld.org, 2013

[2] "The UVM Reference Manual", UVMv1.1, uvmworld.org, 2013

[3] "Application Programming Interfaces", IEEE Standard for SystemVerilog 1800, Feb, 2013