A Survey of Predictor Implementation using High-Level Language Co-simulation

Sean Little
Verus Research, New Mexico
sean.little@verusresearch.net

Abstract—When implementing a testbench for a complex algorithm, the "predictor" component for constrained random verification is often implemented using a low-level language such C or C++ and utilizes the SystemVerilog direct programming interface (DPI). This "golden reference" implementation provides a source of truth and allows for generating an expected result that can be compared with the synthesizable design under test (DUT) outputs in a self-checking testbench utilizing constrained random stimuli.

It is common to model a signal/image processing, cryptographic, or networking algorithms in a high-level language such as MATLAB or Python to prove the concept of the algorithm before porting it to register transfer level (RTL) code. These high-level languages offer many advantages over low-level languages such as C or C++. It is advantageous to reuse this proof-of-concept code directly rather than spending the time and risk of porting this code to both C/C++ for the predictor and to RTL for the DUT. Running high-level code directly from the HDL simulator is sometimes referred to as "co-simulation."

This paper will review some existing approaches for co-simulation including commercial tools from MathWorks and open-source libraries such as CocoTB. We will present a trade space that highlights advantages and disadvantages of existing approaches considering things such as monetary cost, simulation performance, tool support, debugging, and the general usability. Finally, we offer a lightweight, abstract C++ API that can be used with SystemVerilog to allow for co-simulation of arbitrary Python or MATLAB code. We will discuss why this approach has more advantages over existing techniques.

This paper is organized according to the following sections:

I. Modern testbench architecture
II. Example design for illustration
III. Where do we get a good predictor?
IV. Verification framework
V. CocoTB
VI. The HDL Verifier
VII. Introducing a new API for co-simulation
VIII. Performance data and conclusion

Sections I-IV provide background information necessary for introducing the topic of co-simulation. Sections V and VI describe tools commonly used for co-simulation. Section VII introduces a new API developed at Verus Research that allows for co-simulation to work in a much simpler, lower impact way than what is currently possible. Section VIII presents data from an experiment that will compare performance and other data for the different techniques.

I. Modern Testbench Architecture

Modern testbench architects frequently use the concept of constrained random stimuli to achieve good test coverage. Whether using a verification framework such as UVM or something else, constrained random reduces the effort required by the verification engineer to anticipate all the corner cases that must be tested. Rather than explicitly creating test scenarios for every valid corner case, the verification engineer instead rules out stimulus scenarios that represent invalid or unanticipated data. Using random number generators for inputs, these invalid scenarios become "constraints" that eliminate data that does not need to be tested. Often the set of events or scenarios that cannot happen in a system is smaller than the set of events that can happen—and therefore must be tested. Thus, focusing on the smaller set of events—what does not need to be tested—simultaneously reduces the effort of the verification engineer while creating a better testbench that can find more problems early in the development cycle. See Figure **1** for a simplified diagram of a constrained random testbench. Note how a well written testbench includes logic for automatically determining whether the test succeeded or not. It is therefore well suited for automated regression testing because it does not require anyone to manually interpret complex waveforms to look for failures.

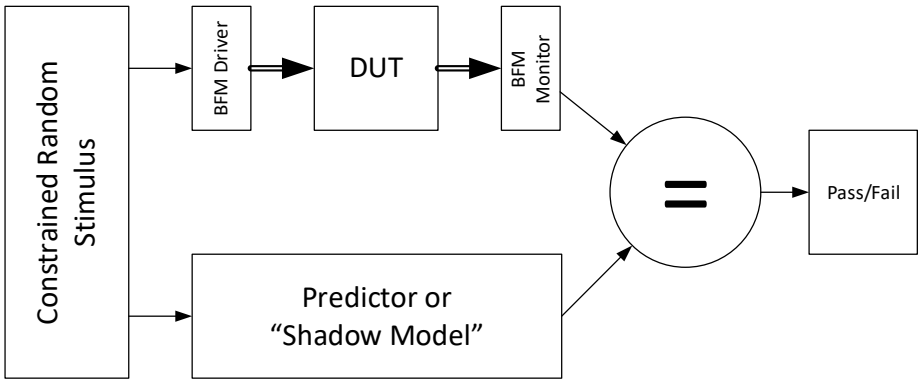

Figure 1: Simplified Constrained Random Test Diagram

Although the advantages of using constrained random stimulus are well-recognized, this approach also comes with its own set of challenges. For some kinds of complex algorithms, it can be challenging to create a comparison in a self-checking testbench. The uncertain nature of the random input data requires a trusted "predictor" that will accurately implement the algorithm in the design under test (DUT). The predictor is also sometimes referred to as the "golden reference" or "shadow model." The implementation of this predictor does not need to be as detailed as the actual implementation in the DUT. For instance, it does not need to be synthesizable. But it must be accurate and, ideally, free from the implementation assumptions made by the DUT designers.

Note that in this discussion we are considering high-complexity designs that implement algorithms in the domains of signal/image processing, cryptography, or networking. A simple design such as a first-in first-out (FIFO) data structure, where the predictor is trivially implemented, would not benefit from the techniques described in this paper.

II. Example Design for Illustration

The example that we will consider for this paper is a polyphase (parallel) finite impulse response (FIR) digital filter. It is often necessary for the implementation of high-rate signal processing algorithms to accept multiple samples per clock. Increasing the width of the pipeline allows for keeping up with data rates that could be multiple giga-samples per second without proportionally increasing the clock frequency driving that pipeline. This is particularly true for FPGAs where the maximum clock rate is limited. The structure of a polyphase FIR filter is illustrated in Figure 2. Note that the filter coefficients or taps are represented by the **h** values fed into the multipliers. Also note this design uses an AXI-streaming interface with full control signals including TVALID and TREADY, even though those details are not displayed in Figure 2.

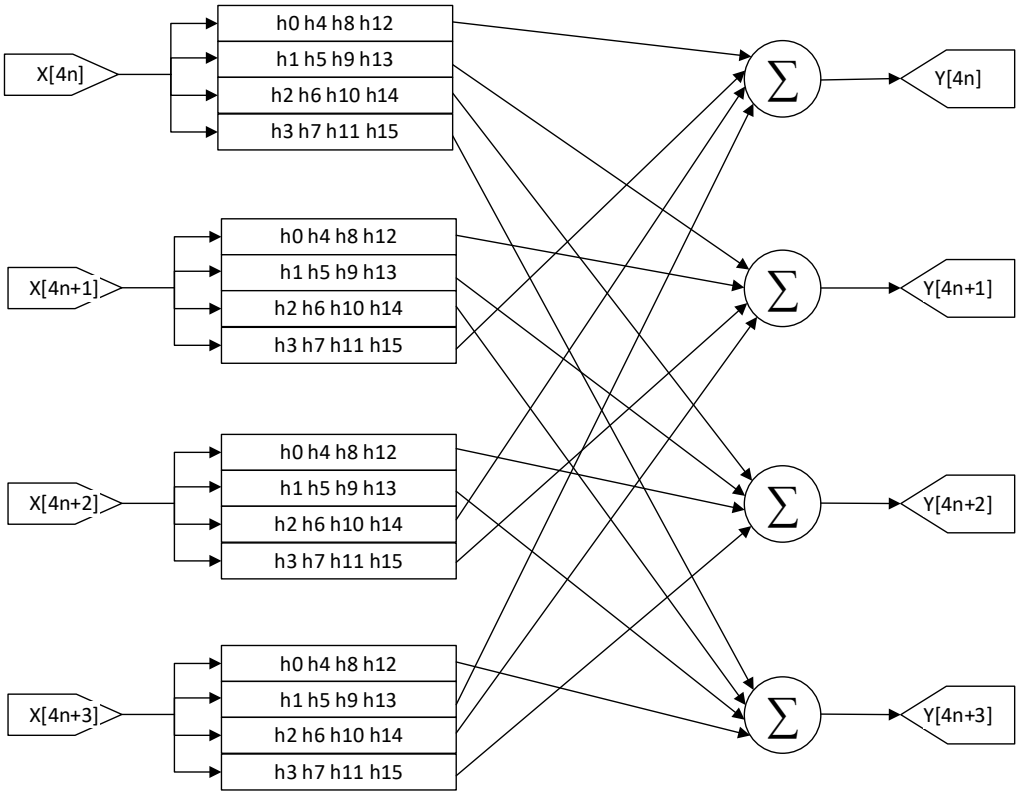

Figure 2: Simplified Polyphase FIR Filter Structure

Utilizing constrained random stimuli is useful for verifying this design. We need to exercise fixed-point corner cases such as overflows and underflows in the input and output data. It is also important to thoroughly exercise the control TVALID and TREADY signals, which should involve driving those signals randomly. Additionally, we may need to build a system simulation that will supply statistically realistic data to exercise the filter and processing downstream from the FIR. We will discuss this in more detail when we discuss the verification framework.

III. WHERE DO WE GET A GOOD PREDICTOR?

Sometimes a predictor is implemented in C or C++ specifically for use in the verification framework. This allows for the testbench foreign language API to interface directly with the predictor code. If the testbench is written in SystemVerilog, for example, the direct program interface (DPI) may be used. If the testbench is in VHDL, one might consider the VHDL procedural interface (VHPI). In either case, the process of implementing the predictor in C or C++ can be quite time consuming and complex, almost to the level of complexity required by the DUT.

Why use a high-level language such as MATLAB or Python for your predictor? Sometimes a high-level model of the algorithm in the DUT already exists and is leveraged by the design team for DUT algorithm prototyping. In this case skipping the step of porting the high-level model to C/C++ is clearly advantageous. But if a predictor must be coded specifically for the testbench, there are numerous advantages and some disadvantages. Languages like Python and MATLAB offer advantages such as large communities of developers, abundant reference material, and high-quality domain-specific libraries. Some disadvantages include possibly slower testbench performance and higher complexity of the build infrastructure. We will explore these trades later in this paper. Note that combining an HDL simulation with a high-level language is sometimes referred to as “co-simulation” implying a cooperation between different languages.

Another advantage of using a high-level language to keep the assumptions between the DUT and test distinct. While industry best practices typically recommend that design and verification teams be split into two distinct teams, this is often not possible. Without a separation between design and verification developers, there is significant risk that the same faulty assumptions might be included in both the design and the test. The test might therefore work perfectly but the design still could have serious problems. One way to provide decoupling of the design and the verification environment is for the test to utilize a third-party library. If a test utilizes constrained random stimulus and the DUT matches an industry standard library according to Figure 1, the developers can have

high confidence that the implementation is functionally correct—even if the development situation differs from industry best practices.

These and other reasons make co-simulation a very powerful tool for some kinds of verification problems. This technique provides a simplified, faster means of implementing a predictor than more traditional methods.

LANGUAGES TO CONSIDER

Unfortunately, there is no API built into SystemVerilog or VHDL that allows for direct co-simulation. Any attempt to use co-simulation must therefore use the foreign APIs provided by the HDL of choice. There are likely other effective techniques for predictor design or co-simulation that is done in different organizations. We are hoping that constructive conversations might result from our sharing what we do, and here are likely good ideas that other people/organizations have on this topic.

This paper will consider the costs and benefits of the following co-simulation libraries:

1. CocoTB [2]
2. The HDL Verifier [1] from MathWorks®
3. An API for co-simulation developed at Verus Research® that supports both Python and MATLAB

Note that for this paper, we are using Siemens QuestaSim® as our HDL simulator. Other simulators that support the SystemVerilog DPI and Xilinx IP core simulation could also be used. We do not have access to tools other than Questa and therefore cannot test them. Unfortunately, open-source simulators do not work for us because we need to run simulations that utilize Xilinx IP and may contain both VHDL and Verilog RTL code. Note also that while there are many useful languages that we could consider, we are going to focus our attention on Python and MATLAB.

A. Python

Python is an interpreted, dynamically typed open-source programming language. Python is an excellent choice for a co-simulation testbench because it features a rich, well documented foreign API for linking the languages to C/C++. There are also many high-quality, actively developed libraries for domain specific algorithm development. Numpy, Scipy, Pycrypto, Scapy, Pandas, etc. are some prominent examples.

The disadvantage with using Python is typically related to performance. This can be a valid concern, but careful consideration of how Python is used in an HDL test framework can virtually eliminate this drawback.

B. MATLAB

MATLAB is a commercial product created by MathWorks. It is also a dynamically typed, interpreted language. It has historically been used for linear algebra, feedback control system prototyping, and signal processing. MATLAB has been around longer than Python, but it has been continuously updated to include some modern constructs such as objects and data structures. There are numerous domain-specific addon products for MATLAB that add capability to the language. One of these is the HDL Verifier that is provided for co-simulation, which we will consider later in this paper. Simulink is another addon product for running simulations in a graphical way, but we will not discuss Simulink in this paper. All MathWorks products have high quality documentation and are actively supported. MATLAB-centric HDL verification is particularly useful when there are teams of people using it in your organization or there is already a large amount of existing MATLAB code.

There are several disadvantages to adopting MATLAB. It is an expensive, licensed, commercial tool and is therefore much less flexible than Python. It cannot be embedded in another application, for example. It also has fewer features such as threading or built-in network constructs. For applications such as networking or cryptography, MathWorks products are probably not a good fit. Performance is also a common concern for a MATLAB based co-simulation verification framework. But again, careful consideration of how MATLAB is used in the framework can go a long way toward improving performance.

IV. VERIFICATION FRAMEWORK

All the testbenches developed and profiled for this paper share the same basic ideas. The FIR filter is being used as a “correlator”. For communications or radar systems the correlator is an important component of the processing pipeline because it can find a predetermined pattern, such as a networking preamble, even in the presence of noise. Randomly inserting the preamble and verifying that the correlator accurately detects it in the expected location is another benefit of using constrained random input data. As stated previously, it is also important to realistically exercise the data path using the fixed-point data representation and show that it agrees sufficiently well with a floating-point model. See Figure 3 for an illustration. You can see the correlator working by producing a red spike exactly where the waveforms were randomly inserted as indicated by the green lines. We want to use MATLAB or Python to generate both the random stimulus and the truth vectors for DUT verification.

For this paper, we are using the Xilinx “FIR Compiler” IP core for the DUT. Using a reliable DUT that is unlikely to have bugs in it allowed us to focus on the verification framework without the need to develop/debug the DUT at

the same time. A detailed discussion of the DUT is beyond the scope of this paper, but we are using the following configuration of the core:

- "Super-sampled" mode of the core with 100 tap coefficients determined in MATLAB and Python
- 10 parallel samples per clock
- Signed 16-bit samples for all input, output and tap data
- 14 fraction bits for the taps, 12 fraction bits for the input, and 5 fractional bits for the output
- 100 micro-seconds of simulation time (with a sample rate of 100Msps)

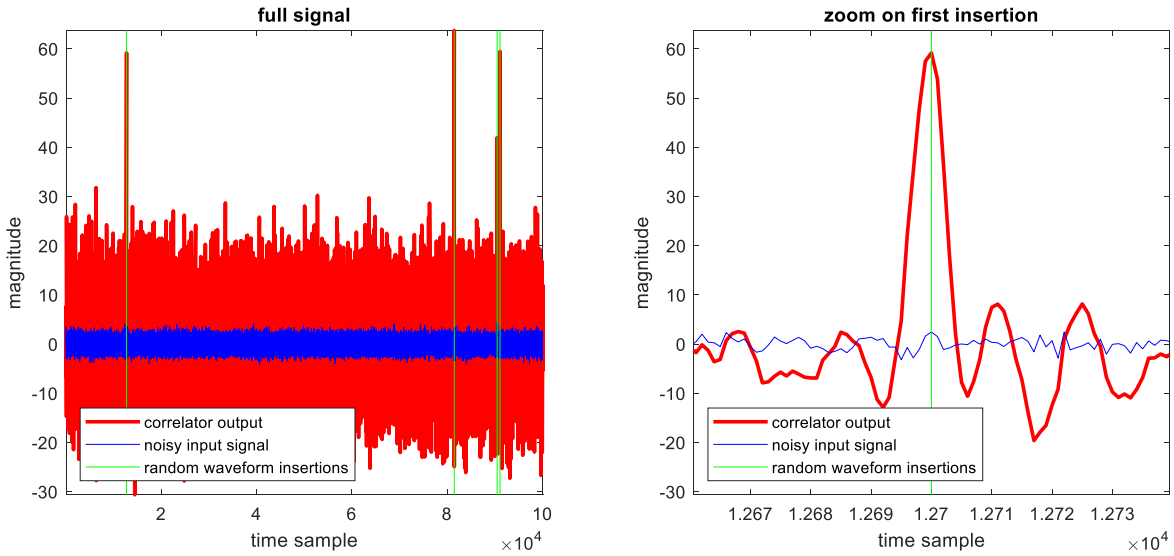

Figure 3: FIR Correlator Example

Note that simulating an IP core can be beneficial even beyond the somewhat contrived requirements for this paper. It is very easy to initialize or drive a complex core like the Xilinx FIR Compiler incorrectly, and simulation is an effective way to find problems caused by doing so. In fact, it took several iterations of testing different configurations of the FIR compiler to get the functionality that we needed for this paper. We should also acknowledge that simulating Xilinx or Altera IP can be challenging since one must consider details such as compiling the simulation libraries, generating code for the core properly, and compiling the IP using the Questasim. We utilized tools for all this, but a discussion of this complexity is also beyond the scope of this paper. Interested parties are welcome to contact us for all the examples that are discussed in this paper.

Also, because we are using AXI-streaming, we need BFMs both to insert the data into the core and to accept output from the core. The co-simulation framework must be able to pass data to and from the BFMs. The specifics of how data is passed to the BFMs is different depending on the co-simulation framework being utilized. Those details will be discussed presently.

V. COROUTINE BASED COSIMULATION TESTBENCH (COCOTB)

CocoTB is one of the more well-known frameworks for co-simulation, built for verifying HDL designs using Python. The philosophy behind this framework is that the entire test architecture is ideally built in Python, with only the DUT implemented in RTL. In addition to the obvious ability to apply the rich Python ecosystem to HDL verification, CocoTB also provides for the following:

- Automatic test generation. Given a list of parameters and other settings, the CocoTB framework will automatically generate tests that exercise all permutations of the specified test parameters. It is easy to build a test that exercises the resets, for example.
- A build system that supports multiple open-source and commercial simulators. Ideally this build system is easily used by automated continuous integration schemes.
- Addon libraries that provide well tested functionality like AXI/AXI-streaming BFMs. These BFMs are very well built and worked well in our testbench.[3]
- Robust support for both Verilog and VHDL.

See Figure 4 for a diagram of our CocoTB test architecture.

As the name implies, CocoTB makes extensive use of Python "Coroutines". A coroutine is an asynchronous Python function that can pause execution and resume later. Every time a python "await" statement is executed, the program switches context to another coroutine that may be ready to "wake up" and resume execution. CocoTB uses coroutines to simulate the parallelism of hardware. You can think of a coroutine await statement in the same way that you would a "@(posedge clk)" event statement in Verilog that might cause execution to switch to another event in the same simulation step. Our testbench uses coroutines to both drive data into the DUT using the BFM driver and to sample data from the DUT using the BFM monitor (see the green shaded blocks in Figure 4). The following code summary illustrates coroutine syntax:

```
async def run_test(dut, idle_inserter=None, backpressure_inserter=None):
    async def drive(dut):
        for test_data in corr.input:
            test_frame = AxiStreamFrame(test_data)
            await tb.source.send(test_frame)

    async def test(dut):
        …

    send_task = cocotb.start_soon(drive(dut))
    test_task = cocotb.start_soon(test(dut))
    await send_task
    await test_task
    …
```

The advertised advantages of CocoTB are also a source of drawbacks. Python is well suited for algorithm implementation, but much less intuitive for modeling concurrent hardware. Here is a list of disadvantages that we encountered when building the CocoTB testbench:

- While writing and debugging coroutines is not difficult, it does require a different mindset than authoring conventional Python code.
- It is possible to use a debugger, but because the Python interpreter is called by a Makefile rather than directly, one must use a remote debugger to pause execution and step through the code. We successfully used the remote debug functionality built-in to Pycharm.
- One must be very careful about things like sign extending data types—things that are not typically considered in Python code. Also, we were unable to find any completely suitable fixed-point libraries. All the open-source Python fixed-point projects we could find appear to be abandoned and lack the features we needed for this test.
- Because the CocoTB scheduler is calling the Python interpreter every clock, one must be very careful to precompute all the stimulus data in advance, or the simulation performance will be very bad.
- It is not easy to use the constrained random solvers that are built-in to SystemVerilog. One must use Python-based random number generation instead. This means that useful features that is already present in SystemVerilog must be rewritten in Python for verification engineers to benefit from infrastructure they are already familiar with.

Other problems not related to using Python for test implementation:

- CocoTB wants to own the whole build system. Indeed, simulating a Xilinx IP core was difficult and required rewriting the underlying Makefile targeting Questasim. Specifically, the CocoTB build process always rebuilds a project from scratch; so, using an external script to initialize the simulation environment by precompiling an IP core is not possible because the Makefiles explicitly delete all previously existing compiled designs and libraries. This framework is not well suited for large projects with hundreds of source files because all files are built from scratch every time the simulation starts.
- The support for Questasim is very rudimentary. There are many things that Questa can do to optimize performance by trading visibility for simulation speed. None of those options are available to the user, and again, rewriting the underlying build infrastructure was required to gain the necessary control of the simulator. The default CocoTB build does not even provide access to the TCL command prompt or to the

waveform viewer. The simulation must be run to completion and the user debugs problems afterward by looking at the logged data files. This means longer turn around times between simulation runs than would be possible in a more conventional workflow.

- There is some evidence to suggest that CocoTB has been successfully run in Windows. But as evidenced by numerous instances of Makefiles directly calling BASH commands (something difficult to support in Windows), CocoTB really does not support a Windows workflow. We were unable to get a Windows build to run in the time available to prepare this paper.

Our summary of our experience of using CocoTB is that, while this framework has some very impressive features, we do not recommend it for large, complex simulations. It is better suited for small, directed unit tests when other options are not readily available.

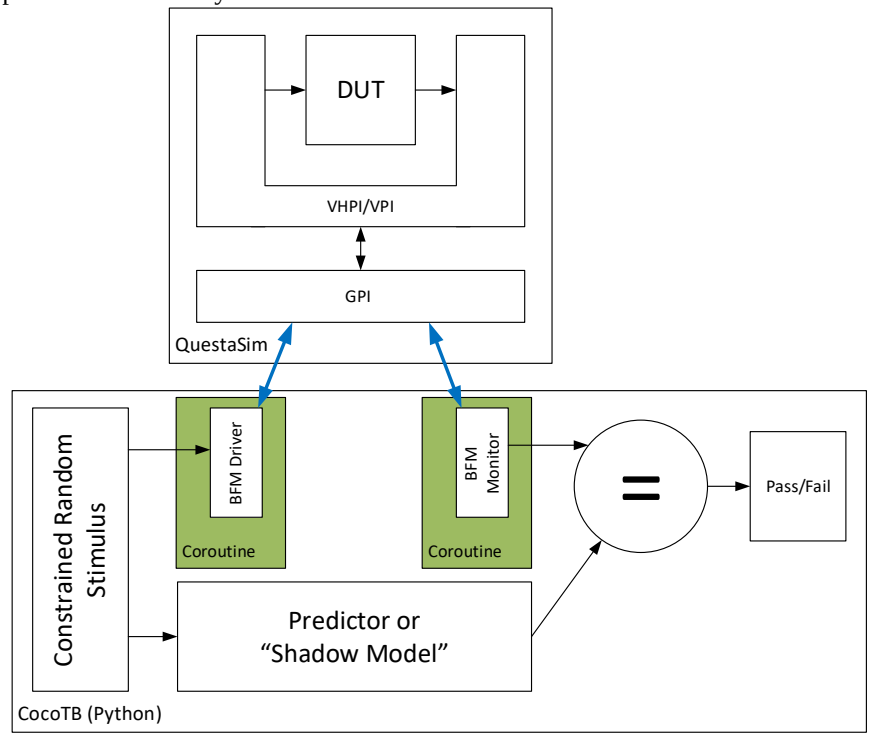

Figure 4: Our CocoTB Architecture

VI. MATLAB Co-simulation using HDL Verifier

As stated previously, the HDL Verifier is a commercial product provided by MathWorks that allows for co-simulation. MATLAB co-simulation assumes that the HDL simulator is the "master" that initiates the simulation, and offloads data to MATLAB to run the predictor. Starting a MATLAB co-simulation session requires initializing the connection between MATLAB and QuestaSim using a special elaboration command. After this initialization, the simulation should be run using the normal QuestaSim workflow. See the diagram in Figure 5.

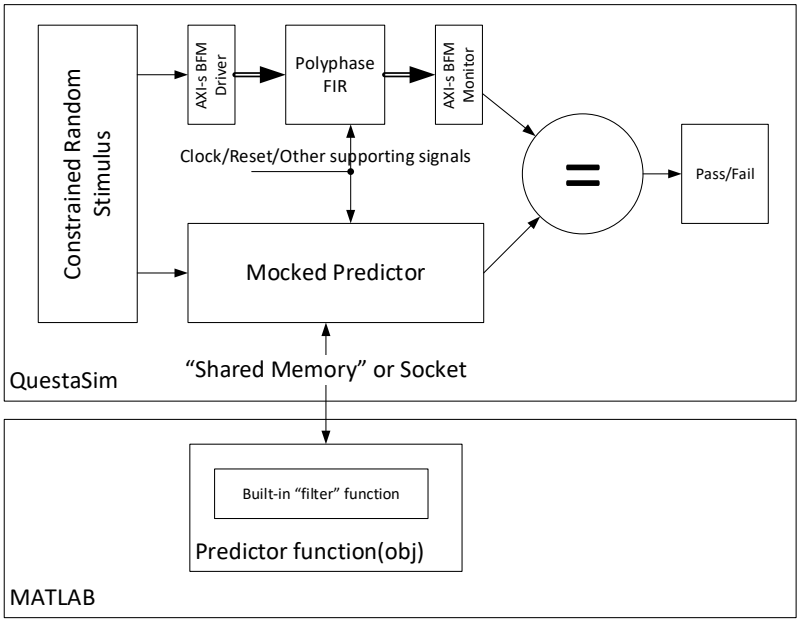

Figure 5: MATLAB Co-simulation Diagrams

The API provided by HDL Verifier is very complex. A MATLAB callback must be specified from the HDL simulation environment that will be called by QuestaSim. This callback code is much more complex than typical MATLAB code, probably because of the many features the API supports. The assumption made by the HDL Verifier is that there is a module in the hierarchy of the HDL design that is substituted by a MATLAB function. The function must therefore sample inputs and drive outputs the way an HDL module would. This is an unnatural way to utilize MATLAB code, but it does provide a lot of flexibility. The MATLAB callback author could, for example, store data that is persistent between calls and update that data utilizing the timing information provided by the API. But if the DUT utilized a bus protocol such as AXI-streaming, the verification engineer would need to implement the logic necessary to sample or drive data from the bus correctly. This task is normally performed by the BFM, and implementing basic BFM functionality might therefore be needed in the MATLAB code. This is not optimal. See **Error! Reference source not found.** for a comparison of advantages and disadvantages of the HDL Verifier.

Advantages	**Disadvantages**
Supports simulation using sockets (computer running HDL simulator might be different from the computer running MATLAB)	Because of the licensed nature of this commercial tool, the MATLAB interpreter cannot be embedded in a separate application. Two applications must therefore run to test one design.
Good port map type support	Non-intuitive, complex workflow. Unusual steps are required to perform elaboration.
Support for VHDL, Verilog and SystemVerilog	Utilization of this tool in a continuous integration (CI) system might be difficult and expensive
Good documentation, examples, and support	Requires non-standard MATLAB code to implement callback functions. Lots of careful following of example code is required to learn the API.
API shares some similarities with FPGA-in-Loop workflows. This may allow for reuse of the verification environment when targeting a supported FPGA development board.	May require BFMs to be reimplemented in MATLAB to drive/sample bus protocols properly.
Support for different simulation configuration options. Tight integration with other MathWorks tools.	Expensive commercial product. No support for open-source simulation tools.
Good support for commercial HDL tool vendors.	

The summary of our experience is that there are many trades associated with adopting the HDL Verifier. While we were not able to complete the example testbench targeting this product in time for publication of this paper, we have used the HDL Verifier for years in different contexts. MathWorks products are very good tools for algorithm development and prototyping. The quality of the documentation they provide surpasses any open-source project that comes to mind, and the quality of their technical support is also very high. The HDL Verifier is a very expensive product that can implement a predictor for a constrained random testbench, but does so in an awkward, needlessly complicated way. It has been our observation that MathWorks suffers from the mindset that all problems should be solved using their products, regardless of whether there are better tools for the job. Like Python, MATLAB is good for many things, but it is not well suited to modeling concurrent hardware. Our recommendation is to avoid the HDL Verifier for co-simulation, and only adopt this product if you need other features that it provides.

VII. HALA: High-level Abstraction Language API

At Verus Research, we have developed an API for sharing data to external languages such that nearly arbitrary functions can be called directly from the HDL simulation tool. We call it High-level Abstraction Language API (HALA). We currently only support SystemVerilog as the verification language since we are using DPI. Support for VHDL could probably be added. The philosophy of this approach is to keep things simple. Rather than supporting a host of esoteric features, we only support calling a function in Python or MATLAB from SystemVerilog. There is no attempt to preempt any functionality already provided by the HDL simulation tool. The external language has no concept of simulation time, for example. Therefore, only functionality analogous to SystemVerilog functions—that do not consume time—can be implemented. HALA is only used for implementing a predictor, and/or providing detailed stimulus data to a BFM. See Figure 6 for a high-level diagram of HALA. We use SystemVerilog for what it is good for, and languages such as MATLAB/Python for what they are good for.

Note that the embedded Python[4] and the MATLAB Engine[5] APIs were used to build HALA.

This allows our example correlator testbench to be very simple and use components that were already available. We have tools to compile Xilinx IP cores for simulation. We have a BFM for correctly stimulating and sampling AXI-streaming interfaces.

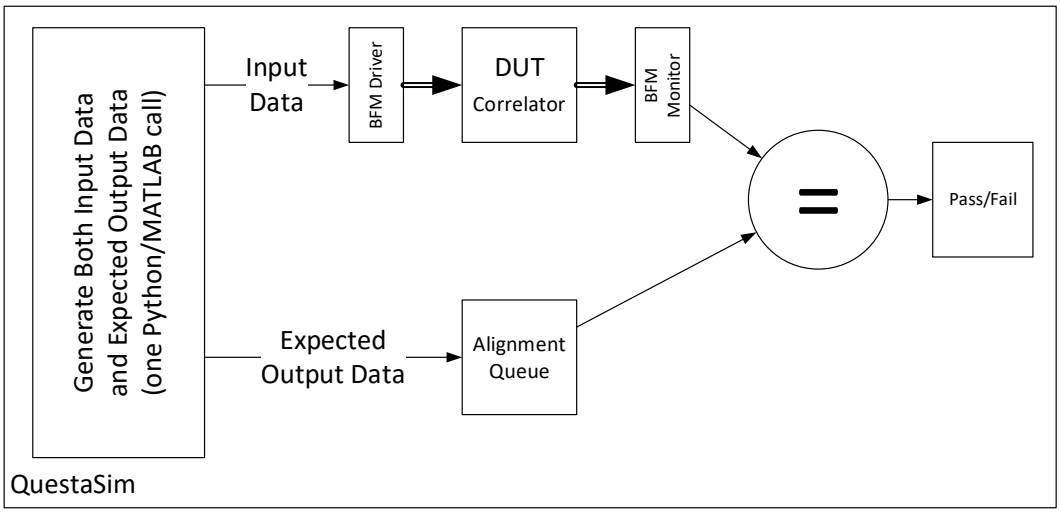

Figure 6: High Level Abstraction Language API (HALA) Test Architecture

Notes about this test:

- We used exactly the same Xilinx IP core for the DUT and test parameters as were used in testing the other frameworks
- The test was implemented using a single MATLAB function call (all conversion to fixed-point was done in MATLAB)
- The test was run in both Windows and Linux
- The Python test of HALA was not completed in time for publication

The following is the essentially the SystemVerilog testbench (the BFM and DUT declarations are not shown):

```
fork
  begin: drive
  cosim_helper_pkg::ints_t input_int, test_int;
  bfm_bases::data_transfer_t input_data;
  matlab_pkg::output_t outputs;
  // pass parameters to MATLAB
  success = success && matlab.run_int("cosim_wrapper", {arg0, arg1, arg2, arg3, arg4},
outputs);
  assert (success || outputs.size() < 2) else
    $fatal(1, "unable to run cosim_wrapper matlab function");
  assert (outputs[0].geti(input_int)) else
    $fatal(1, "unable to convert output[0] into ints");
   input_data = cosim_helper_pkg::Converter#(shortint)::to_bfm(input_int,
SAMPLES_PER_CLOCK);
  num_test_words = input_int.size() / SAMPLES_PER_CLOCK;
  assert (outputs[1].geti(test_int)) else
  $fatal(1, "unable to convert output[1] into ints");
  expected_data = cosim_helper_pkg::Converter#(shortint)::to_parallel(test_int,
SAMPLES_PER_CLOCK);

  drv.put(input_data);
  $info("data insertion complete");
  end

  begin: test
  for (int i = 0; i < num_test_words; i++) begin
    bfm_bases::data_transfer_t outp;
    shortint output_ints[$][$];
    mon.get(outp);
    output_ints = cosim_helper_pkg::Converter#(shortint)::from_bfm(outp,
SAMPLES_PER_CLOCK, '1);
    assert (output_ints[0] == expected_data[i]) else begin
      for (int ii = 0; ii < SAMPLES_PER_CLOCK; ii++) begin
        $display("%0d: %0h != %0h ", ii, output_ints[0][ii], expected_data[i][ii]);
      end
      success = '0;
      $error("mismatch at word %0d", i);
    end
  end
end
join
```

The following tables provide a list of current features and limitations:

Python	
Feature	Limitation
Arbitrary functions can be called from arbitrary locations using relative paths	API should be updated to use a C++ map rather than a SystemVerilog associative array of chandles; all references to chandles could be removed
Virtual environments are supported	Only SystemVerilog “byte”, “longint” and “real” arrays are supported for I/O between Python and SystemVerilog

Arbitrary numbers of inputs and outputs are supported	The run functions only allow for uniform input data. Meaning that multiple inputs can only be one of the supported types. Mixed types are not currently allowed.
Module state is preserved between calls; global data can be updated and used from one call to the next	Functions using integer inputs must be scalars.
Built-in support for remote debugging	
Automated build system that checks for necessary compilers and versions	
Unit tests for both Windows and Linux	
$error, $warning and $info system calls are exposed to python	

MATLAB	
Feature	Limitation
Arbitrary MATLAB functions can be called from arbitrary locations using relative paths	Only SystemVerilog “byte”, “longint” and “real” arrays are supported for I/O between Python and SystemVerilog
Arbitrary numbers of input and output arguments are supported	The run functions only allow for uniform input data. Meaning that multiple inputs can only be one of the supported types. Mixed types are not currently allowed.
API is built on top of a C++ framework that allows any C++ application to communicate with MATLAB (not just HALA)	$error, $warning, and $info system functions cannot be exposed to MATLAB without considerable effort.
Automated build system	Running a MATLAB function that does not have any outputs will currently produce an error
Unit tests for both Windows and Linux	The “finalize” command currently produces a segfault caused by a mismatch between precompiled binaries provided by MathWorks and those provided by Siemens. This is a well understood problem with a current work around that will be fixed given more time.
Works well with fixed-point data (when casted to a supported I/O type)	Requires two memcpy commands to move data from MATLAB to SystemVerilog. This should be reduced to one call.

THE PROMISE OF A UNIVERSAL API

We made the attempt to build a SystemVerilog interface class that would support arbitrary languages for co-simulation. We believe that is possible, but the differences between our targeted languages are significant, and hampered the ability to build a uniform interface. For example, there is no concept of a “module” in MATLAB as there is in Python. Python allows for embedding the interpreter into an external application while MATLAB does not permit this. Also, as we refined the API after implementing it in Python, some things that we initially did using SystemVerilog constructs were moved to C++ data structures for the MATLAB implementation. So, while we have a functional API for both Python and MATLAB, there is still a lot of work to do.

We believe that it is possible to develop HALA into an API that would allow any language with a good C-based foreign interface to connect with SystemVerilog. That was initially the aim of this paper but realizing that goal proved too time consuming to implement in the allotted time frame. Given more time and effort, HALA could include a generic DPI interface that would function like an abstract base-class. If there were a need to implement a SystemVerilog co-simulation bridge to a different language such as R, Julia, or Lua, one could do so by implementing the necessary API defined in that generic interface.

VIII. PERFORMANCE DATA AND CONCLUSION

The following table provides the raw simulation performance data. Specifications of the computers running these experiments:

- Windows 10 computer: Laptop i7-11850H @ 2.50GHz, 128 Gb RAM

- Linux computer: Proxmox VM running Ubuntu 22.04 with generic 64-bit CPU, 64 Gb RAM allocated; same VM used for all experiments

Note that the means for measuring memory were different between CocoTB and HALA. HALA has built-in means for sampling memory, but CocoTB does not have these measurements built-in.

	Total DPI Time (sec)	Total Simulation Time (sec)	Max Physical Memory
HALA MATLAB, Windows	11.9	110	3.8 Gb
HALA MATLAB, Linux	0.06	54	6.6 Gb
CocoTB (Python, Linux)		85.6	584 Mb

The following is a trade matrix for considered techniques:

	CocoTB	HDL Verifier	HALA
Source code provided	yes	no	yes
Direct driving of RTL	yes	yes	no
Build system provided	yes	no	yes
Provides access to simulator	no	yes	yes
Linux support	yes	yes	yes
Windows support	no	yes	yes
MATLAB support	no	yes	yes
Python support	yes	no	yes
What starts the simulation	Makefile	VSIM or Simulink	VSIM

We hope that this has been an illustrative summary of available options for co-simulation. HALA was built to address problems with existing frameworks while not attempting to replace techniques that already work well. But there is a lot of work still to do. We welcome constructive conversations about how other design/verification groups verify complex designs, whether by using constrained random and co-simulation or by using something else. And if the ideas presented in this paper related to a universal co-simulation API have any merit, we are interested in exploring possible means of collaboration.

REFERENCES

[1] (https://www.mathworks.com/help/hdlverifier/hdl-verification-with-cosimulation.html, n.d.)
[2] (https://www.cocotb.org/, n.d.)
[3] (https://github.com/alexforencich/cocotbext-axi, n.d.)
[4] (https://docs.python.org/3/extending/embedding.html, n.d.)
[5] (https://www.mathworks.com/help/matlab/calling-matlab-engine-from-cpp-programs.html, n.d.)