UVM for HLS: An Expedient Approach to the Functional Verification of HLS Designs

Dave Burgoon
Microsoft Corporation
2720 Council Tree Ave., Suite #290
Fort Collins, CO 80525
dave.burgoon@microsoft.com

Robert Havlik
Microsoft Corporation
2720 Council Tree Ave., Suite #290
Fort Collins, CO 80525
rohavlik@microsoft.com

***Abstract*- HLS (High-Level Synthesis) tools allow us to raise the level of abstraction of our hardware design models from RTL (Register Transfer Level) written in Verilog to a much higher, untimed level written in C++. These tools produce Verilog RTL models that are fed to conventional RTL-to-gates synthesis tools, the output of which flow into the physical design process. The advantages of working at this higher level of abstraction are well-documented [1]. These include engineering productivity (e.g. fewer design coding errors), faster time to market, and the ability to quickly modify or leverage a design.**

How are we to directly verify the source HLS model? Unfortunately, our present, unit-level simulation-based functional verification methodology library of choice, the Universal Verification Methodology (UVM) [2], assumes that the model of the DUT (Design Under Test) is written in the same language as the library, namely SystemVerilog/Verilog. This paper summarizes the scenarios we considered for dealing with this situation, and presents an expedient solution based on the UVM Connect [3] open-source library.

I. Introduction

Our team is part of Custom Silicon Development in Microsoft, which has a history of developing custom SoCs, chipsets, and sensors for Xbox, HoloLens, and other devices. When we first started doing HLS design, our verification team focused exclusively on the verification of the synthesized RTL using UVM test benches and tests. This presented some challenges, including

- Achieving sufficient test coverage on the C++ source code.
- Debugging the machine generated RTL.
- Reproducing a given failure in the designer's C++ test bench environment.
- Reconciling a failure observed in RTL simulation with the true design source written in C++.

On our next iteration of HLS design, we decided to pursue a means by which we could *directly* and comprehensively verify the HLS design source, rather than just the generated RTL. We considered several options that fell into these broad categories:

- Single-language solutions
- Emerging commercial EDA solutions
- Mixed-language, co-simulation approaches

This paper summarizes these options in light of considerations about risk, efficacy, nascency, completeness, EDA industry support, and expediency. We then describe our solution which retains the designer's C++ test bench for initial design and testing, then relies primarily on UVM-SystemVerilog coupled with UVM Connect for full verification. This combination enables us to leverage the primary strengths that have been developed in both the C++ and UVM environments, while minimizing duplication of effort. In the designer's C++ test bench, we can leverage a wide range of existing libraries for testing, visualization, debug, and analysis, as well as C++ unit tests to verify and document a block and its sub-blocks while they are being designed. The UVM environment allows us to leverage existing UVM verification IP and expertise, as well as UVM's constrained randomization and coverage measurement capabilities. We also give some details about the implementation of the solution that demonstrate best practices for writing interfaces in the abstracted DUT to enable drop-in RTL replacement, and show how our approach results in a high degree of leverage of test bench and test codes for an RTL-DUT configuration. We then briefly discuss how we solved some unexpected challenges with simulation run-time performance. We close with some observations about the role of "golden reference" models and present some goals for future work.

II. Observations and Concerns

The UVM approach to unit-level testing of RTL designs, depicted in Figure 1, below, relies as much as possible on fairly high levels of abstraction. The individual components of the test bench are implemented using standard object-oriented library (UVM) built on an object-oriented language (SystemVerilog). Stimulus and response data are kept at the TLM (Transaction-Level Modeling) level as much as possible. It is only at the point where drivers and monitors interface with the DUT via virtual interfaces that we are dragged down to the signal-oriented register-transfer level of abstraction. We could instead model the DUT at a TLM level of abstraction using SystemVerilog. Indeed, that was a part of the original promise of the "System" part of that language's moniker. However, this promise never materialized. There are no leading HLS tools we know of that accept design models written in SystemVerilog at a behavioral, TLM level of abstraction. Curiously, C++, or SystemC on top of C++, have emerged as the only languages available to us for modeling HLS designs. There would be no reason to write this paper if this were not the case. It is the mixture of languages, UVM/SystemVerilog and SystemC/C++, and our desire to remain at a TLM level of abstraction[1], that is the core of our problem.

We appear to be at an awkward spot in the evolution of verification methodology and tools for HLS-based design. As we move up in the level of abstraction with which we model our designs, we do not want to be forced to leave behind various verification elements that we have today for RTL designs:

- For RTL design, we can use logical equivalence checker (LEC) tools to formally prove the equivalence of the synthesized gate-level netlist to the RTL source. C++-to-RTL LEC is a difficult problem to solve in general for HLS designs. We know that some in the EDA industry are working on it. When will these commercial LEC tools be available? How well will they work?
- When will tools such as property checkers, model checkers, linters, etc., be available for HLS designs? If they are available today, how well do they work?
- Accellera is working on a SystemC implementation of the UVM [4]. When will this work be complete? Will it become a ratified standard? Will the EDA industry broadly support this standard?
- For RTL design, not only do we have the UVM, but we have a robust set of tools for working with UVM-based test benches and simulations, e.g. debuggers, waveform viewers, coverage management infrastructures, and the like. Will these tools become widely available for HLS-based designs? When?

We will continue to follow the progress of emerging standards such as the SystemC implementation of the UVM, and the Portable Stimulus Standard [5], and the EDA industry's level of embrace and support of these standards. For now, we will rely on our expedient UVM-for-HLS approach to exploit the tools and expertise we have in-hand today.

III. Constraints and Considerations

There were several practical constraints that helped inform our choice of solution.

- As we were ramping down our efforts on the first chip that employed HLS, we had a very narrow window (just a few weeks) in which to investigate improvements for the next chip. We therefore did not have the luxury of thoroughly investigating each and every alternative. We needed something expedient and incremental, with a high degree of immediate availability and relatively low risk of unforeseen surprises. We could not afford to be dependent on commercial solutions that are tied to any one set of EDA vendors or tools, or that are not based on fully-supported, shipping, EDA products.
- To manage risk, we needed a solution that was truly incremental with respect to existing UVM agents that we wanted to reuse. To be able to leverage those agents, we required that we have zero changes to existing (non-HLS) test benches and tests that were clients of those agents.
- To further manage risk, we structured our verification planning so that is was not utterly dependent on a UVM-for-HLS approach; that is, we planned such that, in a worst-case scenario, we could fall back to our traditional RTL-DUT unit testing approach. So, in terms of our project planning, our initial foray into a UVM-for-HLS approach was "extra credit."

The considerations we used to pick our solution were:

- *Efficacy and completeness.* We wanted a solution that was at least as capable as our current RTL-DUT unit testing approach, compromising only on those aspects of testing that are obviously impractical or impossible to test at the HLS level of abstraction. For example, we knew that we would not be able to verify clock gating, clock gate overrides, and external resets with an HLS DUT, as these are not modeled in the HLS source code.
- *Risk and nascency.* To minimize risk, we tried to avoid solutions that are relatively new and unproven.

---

[1] All the leading commercial simulators support mixing Verilog and SystemC modules, but they do so only at the signal level of abstraction.

- *EDA industry support and conformance to existing standards.* So as not to unduly constrain future projects, we wanted the elements of our solution to be as EDA-vendor-neutral as possible, and to be based on current ratified standards rather than draft or emerging standards. We also wanted to be able to avail ourselves of the full portfolio of UVM-SystemVerilog tools from the EDA industry beyond simulators, e.g. debuggers, waveform viewers, and coverage data management systems.
- *Timeliness and expediency.* We wanted a solution that would not require a lot of development, expense, or investigation. We wanted something that was as close to "in-stock" as possible[2].
- *Flexibility and Compatibility.* We needed a solution that can work with both SystemC and pure C++ HLS designs. This allows us to use the best language and abstraction level for a given HLS design.

IV. SOLUTION OPTIONS

We considered the following options for verifying HLS DUTs:

1. Single-language solutions

   a. Eschew SystemVerilog, and use the version of the UVM written in SystemC.

      This option may prove viable in the future, provided that the EDA industry and its customers embrace it fully. This emerging standard [4] is not yet complete, and it remains to be seen if the EDA industry and its customers will rally around it, providing the breadth and depth of solutions that we have today for UVM-SystemVerilog. At this point in time, this option fails with respect to efficacy, risk, EDA industry support, and expediency.

   b. Eschew UVM, and rely on unit-level environments written by the designer in C++ or Python.

      We have found that effective "bring-up" verification can be realized in a test bench written in C++ by the designer using an Integrated Development Environment (IDE) readily available within our company on the PC. However, this approach lacks in the areas of efficacy and completeness for final sign-off verification, as tests still need to be run against the RTL. The tests developed in these environments tend to be directed in nature, and the approach lacks rich constrained randomization and functional coverage measurement facilities.

2. Emerging commercial EDA solutions

   a. Rely on our HLS synthesis tool vendor to supply a solution.

      Besides the problem of being vendor-specific, the solution from our synthesis vendor that we were aware of assumes that the designer's C++ test suite is complete and robust. Assuring this requires constrained-random stimulus generation and coverage measurement capabilities that are at least on a par with the UVM. See 1.b., above.

   b. Use a commercial "portable stimulus" solution.

      The idea here would be to develop of suite of verification tests using one of the languages defined by the draft standard [5], then rely on commercial tools [6] to automatically generate the environment that operates directly on our C++/SystemC DUT. This option may be viable in the near future, but the standard has not yet been ratified nor widely adopted by the EDA industry. Embracing one of the few solutions that are available today would risk us getting tied to a particular vendor. It is also not very expedient, as we would need to climb a learning curve to become proficient in the new language and related tools.

3. Mixed-language, co-simulation approaches

[2] If it's in stock, we got it. ☺

All these approaches share the advantages of efficacy and completeness, as they allow us to leverage our existing UVM-SystemVerilog expertise and IP.

a. Use a vendor-specific co-simulation technology such as Synopsys' TLI (Transaction-Level Interface).

   As mentioned previously, we wanted to avoid vendor-specific solutions.

b. Write our own co-simulation interface library based on the IEEE-1800 DPI standard [7].

   This approach would be neither timely nor expedient, as it would likely entail significant in-house development.

c. Use an open-source, vendor-neutral co-simulation library such as UVM-ML [8] or UVM Connect.

   Given our constraints and criteria, this was the most attractive approach. Both open-source libraries have been around for about five years and are therefore (hopefully) fairly low-risk due to their maturity. They are both built upon the standard DPI and should work well with the leading commercial SystemVerilog simulators. They are also timely and expedient, as they are readily available and well-documented.

   We relied heavily on the excellent overview and insights into these two libraries provided by Long and Aynsley [9] to decide on UVM Connect. It seemed simpler to use than UVM-ML and did not require us to change our style of coding for our C++ components.

V. Implementation

A. Overview

We contrived a simple, small, two-stage, streaming image filter design, called "small_filt," to facilitate our exploration of UVM-for-HLS approaches. The small_filt design is written in untimed C++ and includes common elements and I/O of larger designs, but simple internal functions. The first stage applies a programmable offset to the incoming 8-bit pixel data; the second stage multiplies a programmable scale factor to each pixel and its predecessor, and outputs the average of the two scaled pixels. A simple synchronous bus (not contrived), called "PTB" (for "Pixel Transfer Bus"), transfers one pixel per clock when the "valid" signal is asserted, and is used for small_filt's main input and output. The transaction class (derived from uvm_sequence_item) corresponding to data transfers on this bus was named "ptb_seq_item."

The RTL-DUT version of the small_filt test bench is depicted in Figure 1. Our UVM agents[3], one for the PTB ("ptb_agent") and one for the APB[4] ("apb_agent"), came from an in-house library used by multiple projects. As such, we were restricted to changing their implementation only in ways that were backward compatible. Existing client environments employing these agents needed to continue to work unchanged. The checker component, "small_filt_func_scbd," colloquially called the "scoreboard," incorporates a DPI import of an abstract "golden reference" model written in C++, as is our custom. The scoreboard subscribes to the analysis ports of the PTB agent objects and consults the reference model to determine if the DUT correctly filtered the incoming pixels.

[3] To reduce clutter in the diagram, two other "sideband" agents, one for the reset signal, and one for the clock-gate override signal, are not shown.

[4] AMBA Advanced Peripheral Bus

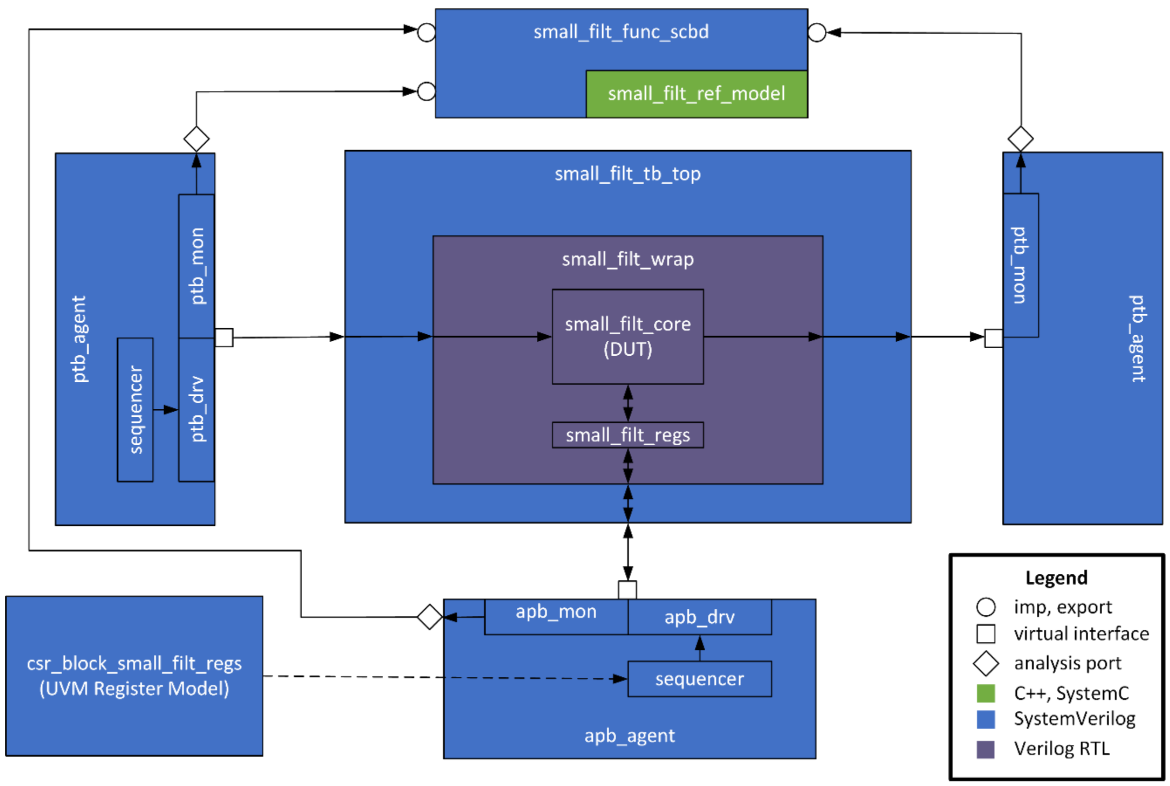

Figure 1. RTL-DUT Test Bench

Our goal was to leverage as much of the test bench, test, and sequence codes as possible between the RTL-DUT environment and the HLS-DUT environment, depicted in Figure 2. Our approach was to leverage the scoreboard as-is, convert from TLM1 sequence items to TLM-2.0 generic payload transactions, develop subclasses for the various UVM environments, agents, and monitors involved, and use type overrides to select the class types needed for the HLS-DUT case. We then interfaced the derived agents to the DUT using TLM-2.0 sockets facilitated by the UVM Connect library and some wrapper code.

In the description that follows, we assume that the reader is well versed in SystemC and the UVM, and has also studied the UVM Connect tutorials and documentation. Tutorial information about SystemC, UVM, and UVM Connect is beyond the scope of this paper.

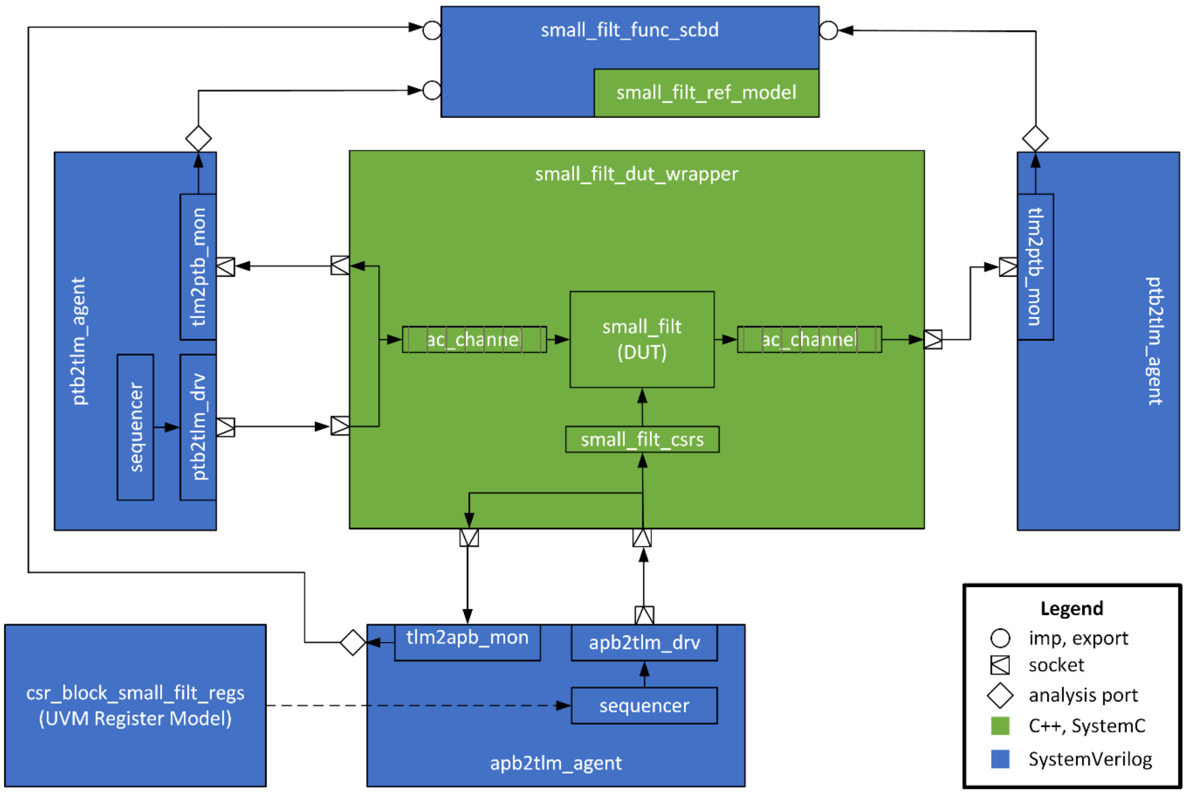

Figure 2. HLS-DUT Test Bench

B. *TLM Conversion*

UVM Connect allows for direct use of TLM1 transactions by providing overrides for the virtual do_pack() and do_unpack() methods of the uvm_sequence_item class on the SystemVerilog side, and template specializations of the uvmc_converter class on the C++ side. However, we thought it would be wiser in the long run to model the transactions crossing the language barrier using TLM-2.0 and the standard "generic payload". Either choice requires writing custom conversion code, but converting to the generic payload enables the possibility of someday plugging HLS DUTs into virtual prototype systems based on SystemC TLM-2.0. It also enables the possibility of better simulation run-time performance via UVM Connect's performance-optimized converter class, uvmc_xl_converter, which exploits TLM-2.0's pass-by-reference semantics (See "Performance Challenges," below.)

One the SystemVerilog side, we augmented our ptb_seq_item transaction class with two new methods:

```
virtual function void to_tlm_gp(ref uvm_tlm_gp txn);
```

and

```
virtual function void set_from_tlm_gp(const ref uvm_tlm_gp in);
```

The to_tlm_gp() method packs the relevant data members of the transaction object into the byte array referenced by the TLM generic payload object "txn." It is used by the driver (ptb_tlm_drv in Figure 2) to convert the ptb_seq_item pulled from the sequencer to a TLM generic payload object that is sent into "tlm_out," a uvm_tlm_b_initiator_socket, via its b_transport() method:

```
task ptb2tlm_drv::run_phase(uvm_phase phase);
  // TLM2 Generic Payload Transaction
  uvm_tlm_gp tlm_gp_req_item = new("tlm_gp_req_item");

  uvm_tlm_time delay = new("delay", 1e-12);

  forever begin
    seq_item_port.get_next_item(req_item);
    req_item.to_tlm_gp(tlm_gp_req_item);
    tlm_out.b_transport(tlm_gp_req_item, delay);

    rsp_item.copy(req_item);
    rsp_item.set_id_info(req_item);
    seq_item_port.item_done(rsp_item);
  end //end forever

endtask: run_phase
```

The set_from_tlm_gp() method is the functional complement of to_tlm_gp(): it sets the relevant data members of the present ptb_seq_item object according to the attributes of the uvm_tlm_gp object “in.” It is used by the b_transport() method of the monitor (tlm2ptb_mon in Figure 2) to create a new ptb_seq_item object to send out its analysis port “mon_items_ap”:

```
// Blocking task called via the "tlm_in" socket
task tlm2ptb_mon::b_transport(uvm_tlm_gp txn_in, uvm_tlm_time delay);
  ptb_txn = ptb_seq_item `PTB_AGENT_PARAMS_PASS::type_id::create("ptb_txn",this);

  ptb_txn.set_from_tlm_gp(txn_in);
  mon_items_ap.write(ptb_txn);

endtask: b_transport
```

There are also conversion methods on the C++ side needed to convert to and from objects of the C++ class corresponding to ptb_seq_item. These are discussed below under “DUT Wrapper.”

C. *Subclasses and Type Overrides*

To maintain compatibility with existing RTL-DUT client test benches and enable the leverage of existing test bench and test codes, we employed a system of subclasses and type overrides. We turned each common agent, driver, and monitor class into a base class from which are derived subclasses specific to the needs of the RTL-DUT and HLS-DUT configurations. Figure 3 depicts the resultant class hierarchy for the PTB-related components, and how we repurposed legacy class names. We turned the legacy ptb_agent into a base class from which ptb2tlm_agent is derived. We repurposed ptb_mon and ptb_drv to be subclasses of ptb_mon_base and ptb_drv_base, respectively, that are used only in the RTL-DUT case. We created new classes tlm2ptb_mon and ptb2tlm_drv that are only used in the HLS-DUT case. Any attributes specific to the RTL-DUT case, e.g. virtual interface handles, were moved into the RTL-DUT subclasses, and any attributes specific to the HLS-DUT case, i.e. the driver’s uvm_tlm_b_initiator_socket and the monitor’s uvm_tlm_b_target_socket, were placed in the HLS-DUT subclasses.

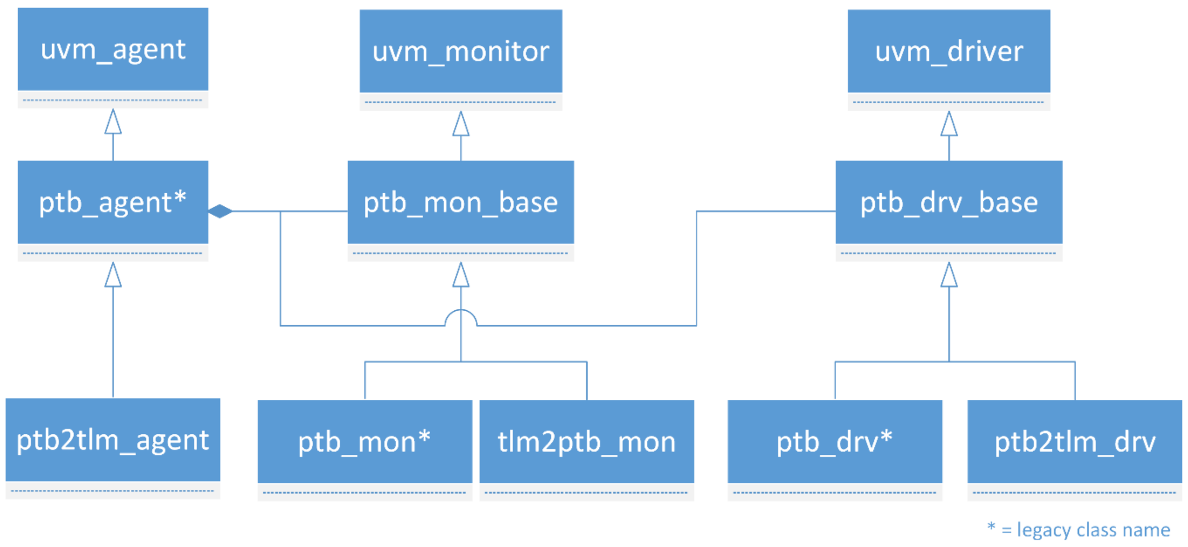

Figure 3. PTB Component Class Hierarchy

We also changed ptb_agent so that it contains monitor and driver base class objects instead of RTL-DUT-specific objects, and used the following code to ensure that legacy RTL-DUT test benches will properly override the base class types:

```
function void ptb_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

  void'(uvm_config_db #(int)::get(this, "", "is_master", is_master));

  if (is_master == UVM_ACTIVE) begin
    sequencer = uvm_sequencer #(ptb_seq_item `PTB_AGENT_PARAMS_PASS)::type_id::create("sequencer", this);

    // By default, create a "ptb_drv" for "driver", but only if an override_type
    // has not been previously set
    ptb_drv_base `PTB_AGENT_PARAMS_PASS::type_id::set_type_override(
      .override_type(ptb_drv `PTB_AGENT_PARAMS_PASS::get_type()),
      .replace(0)
    );
    driver = ptb_drv_base `PTB_AGENT_PARAMS_PASS::type_id::create("driver", this);
  end

  mon_items_ap = new("mon_items_ap", this);

  // By default, create a "ptb_mon" for "monitor", but only if an override_type
  // has not been previously set
  ptb_mon_base `PTB_AGENT_PARAMS_PASS::type_id::set_type_override(
    .override_type(ptb_mon `PTB_AGENT_PARAMS_PASS::get_type()),
    .replace(0)
  );
  monitor = ptb_mon_base `PTB_AGENT_PARAMS_PASS::type_id::create("monitor", this);

endfunction: build_phase
```

Notice the adroit use of the “replace” formal parameter of set_type_override() in the above code. Overriding the default value of 1 with 0 ensures that we respect any type override effected at a higher level in the UVM component hierarchy[5]. In a legacy RTL-DUT test bench, none will exist, so we end up using the appropriate RTL-DUT classes for the monitor and driver.

Moving up in the uvm_component hierarchy, we refactored our uvm_env-derived class into a base class with subclasses, as shown in Figure 4, and employed type overrides to select the appropriate agent subclasses, which, in turn, employ type overrides to select the appropriate driver and monitor subclasses. The cascade of type overrides starts in small_filt_base_test::build_phase(), where we call the appropriate small_filt_base_env type override according to an argument passed in on the simulator command line:

```
void'($value$plusargs("CONF=%s", conf));
if (uvm_is_match("*_rtl_dut", conf)) begin
  small_filt_tb_cfg::is_rtl_dut = 1;
  `uvm_info(REPORT_TAG, "RTL DUT", UVM_NONE)
end
else begin
  small_filt_tb_cfg::is_rtl_dut = 0;
  `uvm_info(REPORT_TAG, "HLS DUT", UVM_NONE)
end

if (small_filt_tb_cfg::is_rtl_dut)
  small_filt_base_env::type_id::set_type_override(small_filt_rtl_env::get_type());
else
  small_filt_base_env::type_id::set_type_override(small_filt_hls_env::get_type());
`uvm_info(REPORT_TAG, "Creating Environment", UVM_MEDIUM)

env = small_filt_base_env::type_id::create("env", this);
```

While it is the environment base class' build_phase() method that constructs the environment object, we rely on the polymorphically chosen build_phase() method of the subclass to effect the requisite type overrides before super.build_phase() is called. For example:

```
virtual function void small_filt_hls_env::build_phase(uvm_phase phase);
  ptb_agent `PIXIN_INF_PASS::type_id::set_type_override(ptb2tlm_agent`PIXIN_INF_PASS::get_type());
  ptb_agent `PIXOUT_INF_PASS::type_id::set_type_override(ptb2tlm_agent`PIXOUT_INF_PASS::get_type());
  apb_env `APB_IF_PARAMS::type_id::set_type_override(apb_hls_env`APB_IF_PARAMS::get_type());
  super.build_phase(phase);
endfunction: build_phase
```

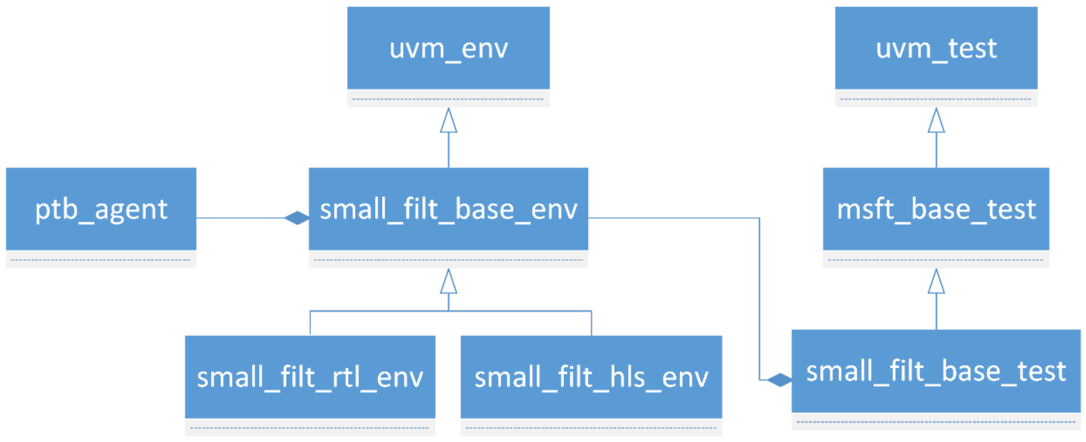

Figure 4. Test Environment Class Hierarchy

[5] Recall that the build_phase() tasks of uvm_components are called in top-down fashion.

D. DUT Wrapper

To interact with the method of the C++ object that is the main processing entry point for the HLS DUT, we need a wrapper that implements the required initiator and target sockets, and implements the b_transport() ("blocking transport") methods that will be registered with the target sockets. Referring to Figure 2, for each agent connected to the DUT wrapper, we need a simple_target_socket, b_transport() method, and simple_initiator_socket:

```
class small_filt_dut_wrapper : public sc_module {

public:
  simple_initiator_socket<small_filt_dut_wrapper> data_in_mon; // defaults to tlm_gp
  simple_initiator_socket<small_filt_dut_wrapper> data_out_mon;
  simple_target_socket<small_filt_dut_wrapper> data_in; // defaults to tlm_gp
  simple_initiator_socket<small_filt_dut_wrapper> csr_in_mon;
  simple_target_socket<small_filt_dut_wrapper> csr_in;

  small_filt_dut_wrapper(sc_module_name nm) :
    data_in_mon("data_in_mon"),
    data_out_mon("data_out_mon"),
    data_in("data_in"),
    csr_in_mon("csr_in_mon"),
    csr_in("csr_in")
  {
    data_in.register_b_transport(this, &small_filt_dut_wrapper::b_transport);
    csr_in.register_b_transport(this, &small_filt_dut_wrapper::csr_b_transport);
    .
    .
    .
  }
  .
  .
  .
private:
  small_filt_ns::small_filt dut;
  small_filt_ns::small_filt_csrs csrs;
  ac_channel<small_filt_ns::ptb_t> ch_ptb_in;
  ac_channel<small_filt_ns::ptb_t> ch_ptb_out;
  .
  .
  .
};
```

Each simple_target_socket receives an incoming transaction and processes it using the b_transport() method registered by the socket. For PTB transactions, our b_transport() method is:

```
virtual void b_transport(tlm_generic_payload& gp, sc_time& t) {
  small_filt_ns::ptb_txn data_in(gp);

  // Tee the incoming transaction into data_in_mon
  data_in_mon->b_transport(gp, t);

  // Send incoming data into the pipeline
  ch_ptb_in.write(data_in);

  // Process any data in ch_ptb_in
  dut.StepAC(ch_ptb_in, csrs, ch_ptb_out);

  // Send output to data_out_mon
  while (ch_ptb_out.available(1)) {
    send_to_monitor(ch_ptb_out.read(), data_out_mon, t);
  }

  // Complete the TLM transaction
  gp.set_response_status(TLM_OK_RESPONSE);
}
```

On the first line above, we initialize "data_in" using a constructor that was added to the ptb_txn class that converts from a TLM generic payload object. Next, we send tlm_generic_payload object "gp" unaltered out the "data_in_mon" simple_initialtor_socket for the sake of the tlm2ptb_mon on the left side of Figure 2. We then write "data_in" into an ac_channel [10] "ch_ptb_in" and call the DUT's StepAC() method, which filters the data in "ch_ptb_in" and places the results in the ac_channel "ch_ptb_out." Finally, we send each item in "ch_ptb_out" into "data_out_mon" for the sake of the tlm2ptb_mon on the right side of Figure 2. The send_to_monitor() method converts an input ptb_txn into a tlm_generic_payload object, and sends it out the designated simple_initiator_socket via a call to the socket's b_transport() method.

Note that all of this happens in zero simulation time; there is no need to modify the incoming sc_time parameter "t" and there is no need to yield by calling sc_core::wait(). For our streaming data-driven DUT, this overhead is simply not needed. As pixels come in to the DUT wrapper, they are immediately placed into the input queue of the scoreboard via a monitor. The filtered output is computed in zero time and immediately placed on the scoreboard's output queue, again via a monitor. So, from the scoreboard's point of view, all inputs and actual outputs appear in the correct order, with the actual outputs suitable for comparison with expected outputs as computed from the DPI-imported golden reference model.

The other blocking transport method of small_filt_dut_wrapper, csr_b_transport(), services APB transactions that effect changes in the control/status register object "csrs" passed into StepAC().

E. UVM Connect "Wiring"

The last piece of our implementation that we describe is how we "wired" the UVM drivers and monitors to the small_filt_dut_wrapper's sockets using UVM Connect. The SystemVerilog side of the connection is implemented in the ptb2tlm_agent as follows, using the uvmc_tlm::connect() method:

```
function void ptb2tlm_agent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  if (is_master == UVM_ACTIVE) begin
    uvmc_tlm #(uvm_tlm_gp,
               uvm_tlm_phase_e,
               uvmc_xl_tlm_gp_converter)::connect(driver.get_tlm_out(), driver.get_full_name());
  end

  uvmc_tlm #(uvm_tlm_gp,
             uvm_tlm_phase_e,
             uvmc_xl_tlm_gp_converter)::connect(monitor.get_tlm_in(), monitor.get_full_name());

endfunction: connect_phase
```

The first formal parameter of uvmc_tlm::connect() is the handle to the socket object. Notice that we must use a virtual accessor method, e.g. ptb2tlm_drv::get_tlm_out(), to supply the actual parameter, because “driver” is declared to be a handle for the base-class type ptb_drv_base, and only the derived class ptb2tlm_drv actually contains the socket object “tlm_out”. The virtual accessor methods are defined so that the base-class implementation issues a fatal error, and the HLS-DUT derived class implementation returns the socket handle:

```
function uvmc_tlm #(uvm_tlm_gp,
                    uvm_tlm_phase_e,
                    uvmc_xl_tlm_gp_converter)::port_type ptb_drv_base::get_tlm_out();
  // This function definition should never be called.
  // Only sub-class definitions of this function should be called.
  `uvm_fatal(REPORT_TAG, "Base virtual function should never be called.")
  return null;
endfunction: get_tlm_out

function uvmc_tlm #(uvm_tlm_gp,
  uvm_tlm_phase_e,
  uvmc_xl_tlm_gp_converter)::port_type ptb2tlm_drv::get_tlm_out();
  return tlm_out;
endfunction: get_tlm_out
```

The second formal parameter of uvmc_tlm::connect() is the string that is used by UVM Connect as the key on the C++-side to look up the socket. We use uvm_component::get_full_name() to supply the actual parameter to ensure that look-up keys are unique and discernable.

The C++-side of the wiring story occurs in the sc_main() function (not shown in Figure 2):

```
int sc_main(int argc, char* argv[]) {
  small_filt_dut_wrapper& dut_wrapper(*(new small_filt_dut_wrapper("dut_wrapper")));

  uvmc::uvmc_connect<uvmc_xl_converter<tlm_generic_payload> >(
    dut_wrapper.data_in,
    "uvm_test_top.env.pixin_agent.driver"
);
  uvmc::uvmc_connect<uvmc_xl_converter<tlm_generic_payload> >(
    dut_wrapper.data_in_mon,
    "uvm_test_top.env.pixin_agent.monitor"
  );
  uvmc::uvmc_connect<uvmc_xl_converter<tlm_generic_payload> >(
    dut_wrapper.data_out_mon,
    "uvm_test_top.env.pixout_agent.monitor"
  );

  uvmc::uvmc_connect<uvmc_xl_converter<tlm_generic_payload> >(
    dut_wrapper.csr_in,
    "uvm_test_top.env.apb_master_env.agent[0].driver"
  );
  uvmc::uvmc_connect<uvmc_xl_converter<tlm_generic_payload> >(
    dut_wrapper.csr_in_mon,
    "uvm_test_top.env.apb_master_env.agent[0].monitor"
  );

  sc_start();
  return 0;
}
```

VI. Performance Challenges

Recently, we applied what we learned from our contrived "small_filt" example to real design case destined for a chip currently under development. This "big_filt" design involved more, non-trivial stages of processing, and an additional type of agent beyond the pixel and APB agents. After we got the flexible test bench put together, we noticed that the HLS-DUT configuration ran about 8 times slower than RTL-DUT configuration! We did not expect that an untimed, data-driven C++ model in which all the interesting processing happens in zero simulation time would be slower than an event-driven RTL model that takes many cycles to complete its processing.

We found that there were three main reasons for this:

1. Naïve use of the UVMC_INFO macro.

   UVM Connect provides handy facilities for sending messages from C++ to UVM's report server. We had some invocations of the UVMC_INFO macro in our DUT wrapper code, and we used sc_object::name() as the actual parameter for the "context" formal parameter. This caused a lot of time to be spent in the uvm_is_match() function by UVM Connect code that was searching the SystemVerilog-side hierarchy in vain for our SystemC object. By instead supplying the null string as the actual parameter, we found that the underlying code foregoes the search, and our performance improved such that the HLS-DUT configuration was about 2 times slower.

2. Use of the default UVM Connect converter classes.

   Originally, we did not supply any explicit template parameters in our uvmc_tlm::connect() and uvmc::uvmc_connect() calls, which caused the underlying code to the use default converter policy classes. These converters rely on implementations of the virtual do_pack() and do_unpack() methods of uvm_object that employ brute-force unconditional data copying to effect transfers across the SystemVerilog-C++ language barrier. A much more efficient approach, employed by UVM Connect's "XL" converter classes, is to exploit the TLM-2.0 protocol's pass-by-reference approach for the data payload, and the rules around the modifiability of transaction attributes [11] to avoid unnecessary transfers. Once we fixed our "wiring" calls to make use of the uvmc_xl_tlm_gp_converter on the SystemVerilog side and uvmc_xl_converter on the C++ side (as shown in the above code snippets), we enjoyed a significant speed-up: the HLS-DUT case was now about 3 times faster than the RTL-DUT case.

3. Use of debuggable, non-optimized C++ code.

   Once we changed our g++ compiler option from "-g -O0," which results in debuggable, non-optimized code, to "-O3," we enjoyed an additional speed-up factor. Our HLS-DUT case is now about 4 times faster than the RTL-DUT case.

VII. Golden Reference Models

In this section, we consider a common scenario in which HLS is used to design custom hardware that accelerates an existing software algorithm. This scenario often requires the following source codes to be developed to map an algorithm into gates:

1. A software reference model, usually written in a high-level language (e.g. C++, C#, Python, etc.).
2. A fixed-point hardware model that models hardware approximations (e.g. fixed-point computations, memories, configuration registers, etc.). It is used to validate the results of the hardware approximations.
3. An RTL model, written in a hardware description language (e.g. Verilog or VHDL) that implements the fixed-point hardware model at the register-transfer level.

It is possible that, under certain constraints, the software reference model may be directly used by the scoreboard as the validation model for the RTL, allowing the development of the fixed-point hardware model to be skipped. However, we have seen that this is often only the case for relatively simple designs where the validity of an output can be constrained within some tolerance of the software reference model. Some other examples that commonly drive the need for a fixed-point model include the following:

- The need to validate sequential hardware approximations.

- Compatibility with calls to imported DPI functions and the associated hardware configuration settings.
- Insufficient outputs for verification, for example transaction-level outputs and intermediate outputs.

Figure 5, below, shows the flow for a conventional verification process for a hardware accelerator RTL design.

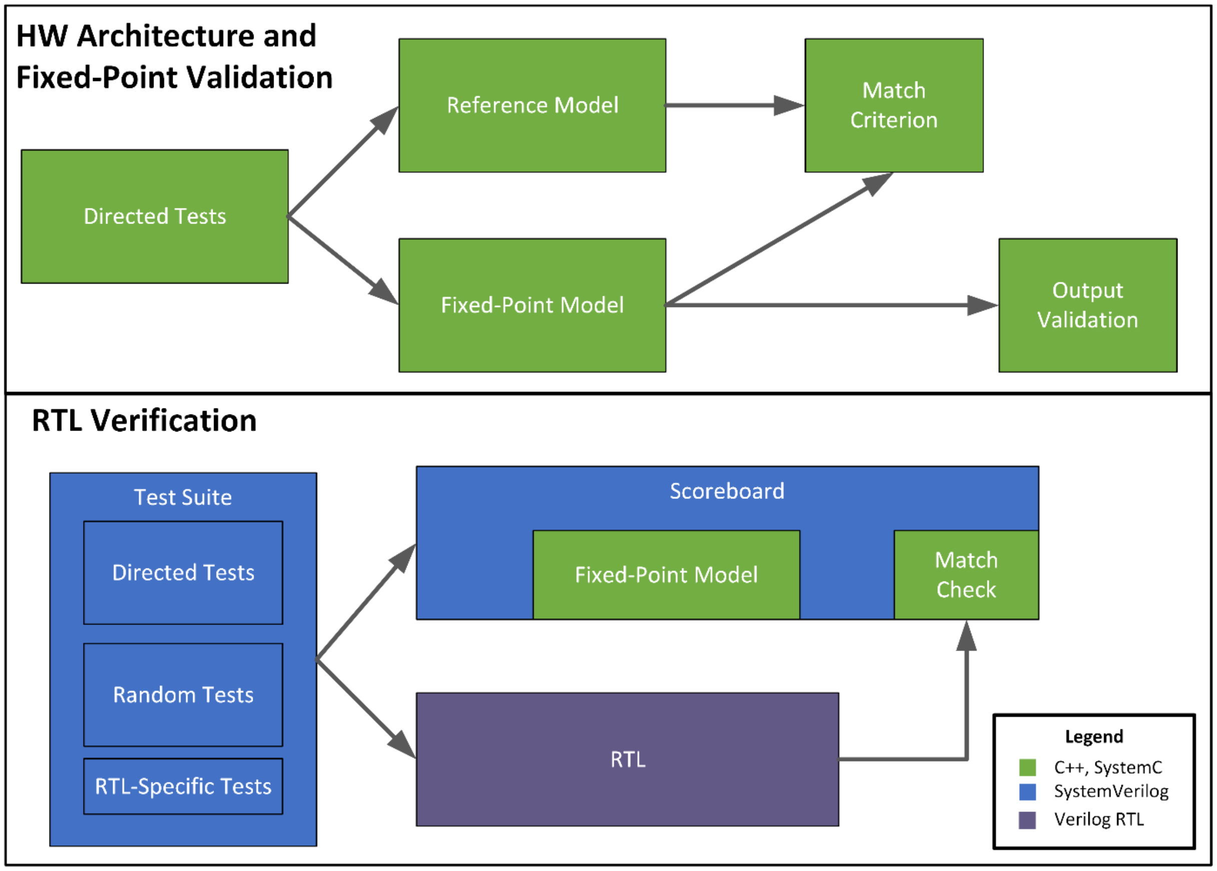

Figure 5. Conventional Three-Source Hardware Accelerator Verification

In this three-source verification flow, the software reference model is generally the true "golden" model, as it is written in a higher-level language and has generally undergone significant application testing. But when it comes to RTL verification in the three-source flow, the assumption is that the fixed-point model is proven to be a "golden" reference model in the context of RTL testing. In many designs, this creates an unseen verification hole in that the UVM-based testing for the RTL will generally push the design into configuration input spaces beyond the bounds of the testing versus the software reference model. When this happens, mismatches with the RTL need to be reconciled and potentially re-validated against the reference model, or both the RTL and the fixed-point model may be producing undesirable results that are not detected.

Using our UVM-for-HLS approach, the verification flow can effectively be reduced to two design sources—the reference model remains the same and HLS source becomes the fixed-point model, as depicted in the upper half of Figure 6. UVM-for-HLS enables us to bring the advanced constrained-random stimulus capability of UVM to target testing of the software reference and HLS fixed-point models. This lifts the sophisticated randomized testing to directly test the hardware approximations, which can often be the most important equivalency to verify.

Referring to the lower half of Figure 6, after the HLS design is sufficiently verified versus the software reference model, the UVM-for-HLS test suite can be directly re-used for testing the RTL that is synthesized from the HLS source code. The present state of art in HLS technology does not guarantee the logical equivalence of the generated RTL for all designs and design styles (as is the case with RTL-to-gate synthesis), so RTL testing and coverage closure are still generally required. But now, in the context of the two-source verification flow with UVM-for-HLS, the fixed point (HLS) model has been sufficiently verified and can be used as a reference against the generated RTL. As the HLS tools strive to produce equivalent RTL, this reduces RTL testing into an exercise of closing coverage and testing RTL-specific features, like resets, clock gating, etc., via the "RTL-Specific Tests" depicted in the lower half of Figure 6. As a bonus, the RTL coverage closure is likely to require running only a subset of tests from the full "UVM Test Suite," greatly reducing the time spent simulating with an RTL DUT, and focusing the bulk of testing cycles on the fast C++/SystemC models.

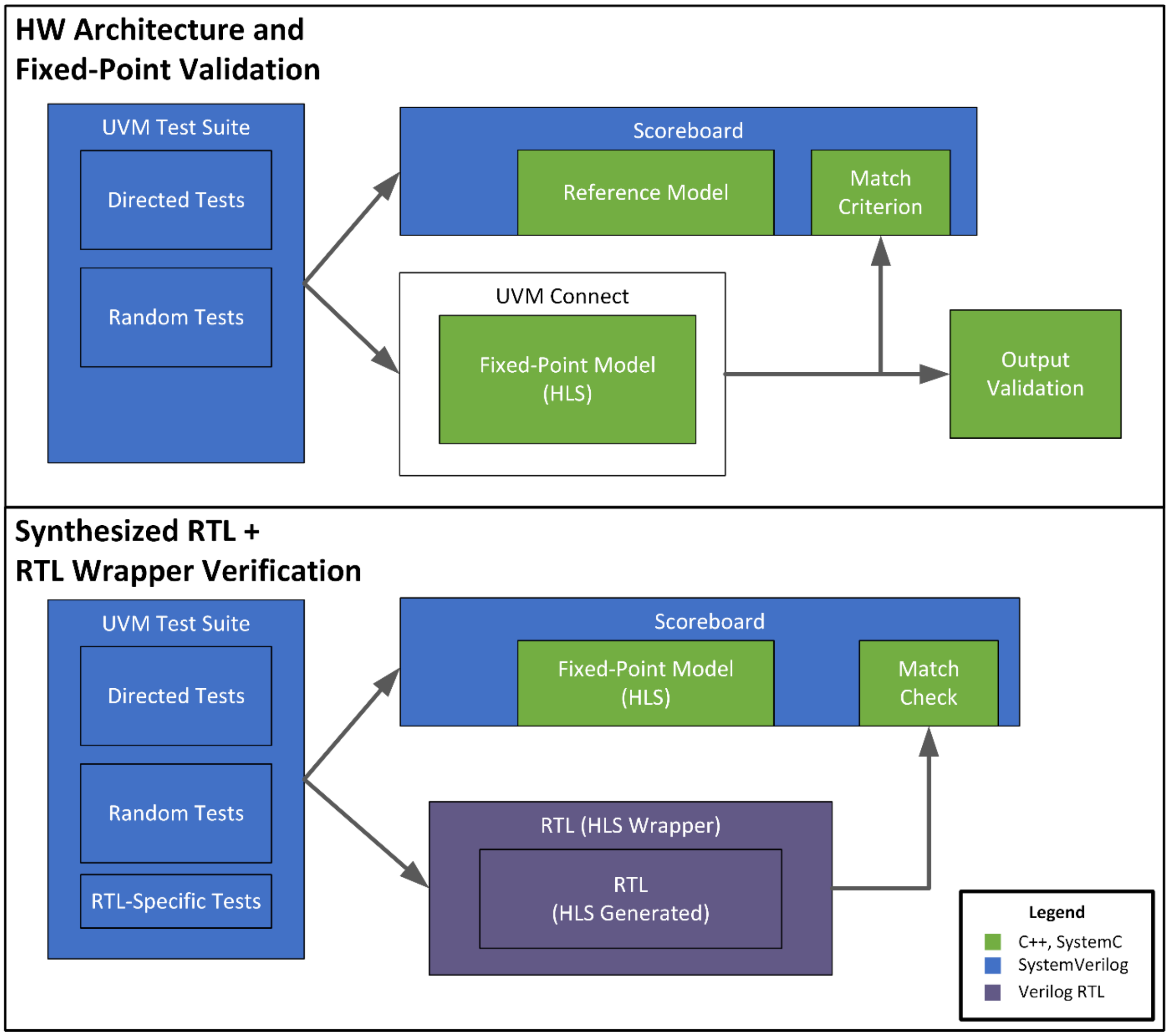

Figure 6: Two-Source Hardware Accelerator Verification via UVM-for-HLS

VIII. Future Opportunities

As we look to future, there are some opportunities to improve and enhance our UVM-for-HLS methodology:

- Continue to improve the simulation run-time performance of the HLS-DUT configuration. Explore opportunities to improve the UVM Connect library to more aggressively employ pass-by-reference semantics, and eliminate data copying as much as possible. Work with the keeper of UVM Connect to incorporate our improvements.
- Develop base classes and standard templates so that our in-house UVM test bench code generator system can accommodate our UVM-for-HLS approach.
- Look for ways to reduce the effort required to adapt the published software programming model for control/status registers and memories to the more abstract model that is often encapsulated in our HLS code. For some designs, we found ourselves writing adapter code multiple times, i.e. in the imported DPI wrapper functions employed by the scoreboard, and in the HLS-DUT wrapper. Is there a way to automate or eliminate the production of this code?
- Explore ways, e.g. portable stimulus tools, to seamlessly leverage test cases across environments and languages. Any test should only need to be written once and in one language. For example, we would like to be able to easily leverage the HLS designer's bring-up test suite written in C++ so that it can run on our UVM-for-HLS test bench.

IX. Summary

Rather than wait for the EDA industry to deliver complete, robust solutions for the functional verification of HLS designs, we pursued an expedient approach that uses UVM Connect to leverage existing industry standards and their attendant commercially available solutions, as well as our own in-house expertise with the UVM. By employing class inheritance and type overrides, we confined the incremental effort needed to directly verify the HLS design to the development of HLS-DUT-specific subclasses and TLM conversion methods, and a SystemC wrapper sc_module. This allowed us to get an early start on the development of unit-level HLS-DUT test bench environments and tests. When it came time to close coverage on the synthesized RTL and verify RTL-specific features, we enjoyed a high degree of leverage: our base classes, the vast majority of the test cases, and the scoreboard in its entirety, were directly reused without change.

Acknowledgments

The authors would like to thank our management team, Mike Erickson and Boris Bobrov in particular, for their support and encouragement in the pursuit of HLS methodology and tools in general, and our UVM-for-HLS verification approach in particular.

References

[1] P. Coussy and A. Morawiec (Eds.), High-Level Synthesis from Algorithm to Digital Circuit, Springer, 2008.
[2] IEEE Std. 1800.2-2017, *IEEE Standard for Universal Verification Methodology Language Reference Manual*.
[3] Mentor, a Siemens business, *UVM Connect and TLM-2.0 Primer*, 2015, https://verificationacademy.com/verification-methodology-reference/uvmc-2.3.1/docs/html/index.html.
[4] Accellera Systems Initiative, *UVM-SystemC Language Reference Manual*, draft, December 2015, http://www.accellera.org/images/downloads/drafts-review/uvm-systemc-1.0-alpha1.tar.gz
[5] Accellera Systems Initiative, *PSS Early Adopter (EA) Portable Stimulus Standard*, June 2017, http://www.accellera.org/images/downloads/drafts-review/PSS_Early_Adopter_Release.pdf
[6] Nguyen Le and Mike Andrews, "Efficient Bug-Hunting Techniques Using Graph-Based Stimulus Models," *Proceedings of DVCon 2016*, http://events.dvcon.org/2016/proceedings/papers/05_3.pdf.
[7] IEEE Std. 1800-2012, *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, Annex H.
[8] *UVM-ML Open Architecture*, June 2017, http://forums.accellera.org/files/file/65-uvm-ml-open-architecture
[9] D. Long and J. Aynsley, "UVM and SystemC Transactions—An Update," *Proceedings of DVCon 2016*, http://events.dvcon.org/2016/proceedings/papers/13_1.pdf
[10] Mentor, a Siemens business, *Algorithmic C (AC) Datatypes*, June 2016, https://www.mentor.com/hls-lp/downloads/ac-datatypes
[11] IEEE Std. 1666-2011, *IEEE Standard for Standard SystemC® Language Reference Manual*.