Is Specman Still Relevant? Using UVM-ML to Take Advantage of Multiple Verification Languages

Timothy Pertuit[1] Doug Gibson[2] David Lacey[3]
[1]Hewlett Packard Enterprise, 5400 Legacy, Plano, TX 75024, 972-605-3192, Timothy.Pertuit@hpe.com. [2]Hewlett Packard Enterprise, 3404 E Harmony Road, Fort Collins, CO 80528, 970-898-2639, Doug.Gibson@hpe.com.
[3]Hewlett Packard Enterprise, 3404 E Harmony Road, Fort Collins, CO 80528, 970-898-7347, David.Lacey@hpe.com.

***Abstract*- Introduction: Silicon Design Lab (SDL) has a long history of developing and using industry leading verification methodologies. In the past, SDL utilized a proprietary methodology based on TestBuilder and SystemC. More recently, we have built our simulation environments with UVM, utilizing both Specman/e and SystemVerilog verification languages. Through the years, we have seen the need to have a flexible environment which supports multiple verification languages all working together. As a result, we have adopted the Accellera UVM multi-language (UVM-ML) approach within our organization. With this approach, we are well equipped to use powerful verification languages such as Specman/e, utilize a broad range of verification collateral developed in UVM-SV, and support partner and industry engagements with other companies utilizing verification collateral based on SystemC and C++.**

UVM-ML is widely adopted across the industry, companies, and EDA making VLSI teams well served to understand its use and capability. This paper will provide insight into how we adopted UVM-ML, will describe a side-by-side example of a verification component that was written in both UVM-SV and UVM-e, and will showcase how UVM-ML has equipped us to easily adapt to a rapidly changing environment including the enablement of emerging industry standard interfaces.

I. Introduction

In today's verification landscape, teams must use a wide variety of technologies to verify the increasingly complex designs. Teams must make use of VIP developed from a wide range of sources from internally developed VIP and purchased 3rd party VIP, to content from partner organizations. With all of these sources and the long history of verification technology, these VIPs may come in many languages and many forms. UVM, specifically UVM-SV, has recently become the de facto standard for VIP in the industry, but there are still many VIPs that were and are developed using Specman/e or SystemC, among others. This paper will focus on case studies involving the integration and interoperability of VIPs from these three languages, Specman/e, SystemVerilog, and SystemC.

A. What is UVM-ML

What exactly is UVM-ML? As background, UVM itself is a standard that aims to improve the interoperability of VIP along with reducing the cost of repurchasing or rewriting IP for each new project [1]. Included in the methodology is a standard class reference implemented in SystemVerilog (referred to in this paper as UVM-SV). Other languages have also implemented part or all of the methodology including Specman/e (UVM-e) and SystemC (UVM-SC). To support the interoperability goal of UVM, a framework is needed to facilitate the mixing of these languages as teams combine VIPs from various sources. UVM-ML provides this framework in the form of an open source library that enables the integration of components written in multiple languages.

B. The Need for UVM-ML

SystemVerilog and the UVM-SV implementation continues to evolve and provide a full featured capable verification language and environment. It has become the de facto standard in many instances due to multi-vendor simulator support and widespread adoption. Specman/e has an even longer success history and continues to offer advantages over SystemVerilog as a verification oriented language. Specman/e code is often fewer lines of code than the equivalent for SV which will typically lead to fewer bugs and easier debug. Specman/e also offers features such as aspect oriented functionality and constraints from above (CFA) which enables a more flexible method for writing test sequences. SystemC often fills the role of high level C models of a DUT which can also be useful as a verification component. As natively compiled code it can offer the highest performance, but is limited in some ways as a full verification language.

Each of the languages supported by UVM-ML has its own advantages and disadvantages. Companies need the ability to choose the language that best fits their needs. With UVM-ML teams can choose the best language for each component depending on that component's requirements. Choosing the right language for the job now becomes possible at a much lower level. Teams are not required to choose the same language for all components in a system, allowing the language to vary depending on the use cases and requirements of the specific component.

C. Case Study Overview

This paper includes case studies and learnings from projects within the Silicon Design Lab (SDL) of Hewlett Packard Enterprise (HPE). The cases explored in this paper all start from a long history and strong foundation of Specman/e and UVM-e test benches and tests. This paper will primarily be looking at the porting of UVM-e content to UVM-SV and the integration of UVM-SV content within the context of UVM-e test benches. Many of the concepts should apply to cases of UVM-e/UVM-SC content integrated with UVM-SV test benches, but the specifics are beyond the scope of this paper.

SDL recently undertook the task of enabling VIP in the industry for a new developing industry standard. However the existing internally developed VIP was entirely UVM-e based. In order to provide the most value and best interoperability with other companies, it was decided that some of the VIP components should be ported to UVM-SV. By taking advantage of the UVM-ML libraries, the new components became the primary VIP for both internal and external use cases. Existing test benches and tests were able to be reused almost entirely with little disruption as UVM-ML provided a complete integration of the components. Through the use of proxies on the UVM-e side most of the multi-language implementation details are hidden from the test writer. This use case will look at the porting of multiple complex interface verification components (VCs), including layering of components to provide high levels of reuse between test benches.

II. UVM-ML USES, CAPABILITIES, BENEFITS

A. Fundamentals of UVM-ML

Terminology

UVM-e and UVM-SV have slight differences in terminology for the same components. This paper will use the UVM-SV names of those components for consistency regardless of the language being discussed. Specifically this paper will use 'sequencer' instead of 'sequence driver' to mean the component that sequences are executed on. This paper uses 'driver' instead of 'bfm' to indicate the component that drives the wires of the device under test, DUT.

Multilanguage Data Communication

The foundation of the UVM-ML framework is in providing a mechanism for creating communication channels between various languages. This is accomplished primarily through the use of TLM ports. UVM-ML provides the backend components necessary for creating connections between TLM ports of different languages. UVM-ML supports both TLM 1.0 and TLM 2.0 including both blocking and non-blocking versions of TLM ports providing significant flexibility in the types of functions and tasks that can be exported across the languages.

The TLM ports form the physical channels, but in order to provide a full communication framework there must be a mechanism of converting the data types of one language to the others. This is accomplished through the use of equivalently defined types in all the associated languages as well as data serialization and de-serialization routines for representing the content into a consistent form. Creating data types that are equivalent in multiple languages is a necessary step in this process and can be done manually or with the help of automation tools such as the `mltypemap` tool that is provided as part of the Cadence Incisive simulator. Over the course of several projects SDL successfully used both the manual and automated methods of type mapping.

In addition to type mapping, there is the need for data serialization routines. Data serialization refers to the packing of a `struct` or `class` in a consistent bit stream from language to language. Data serialization is very similar to the mechanisms used for packing fields of `uvm_objects` using the `uvm_packer`. In some cases the automatic field packing provided by the UVM field macros can produce the necessary data serialization. It is however also possible to create separate data serialization classes that handle the necessary transformations so that any packing done for UVM-ML is separate from what may be needed internally within the components themselves. The data serialization routines can be manually written in much the same way as one would manually write the field packing routines of an `uvm_object`, or can be automated using the same `mltypemap` utility mentioned previously.

UVM-ML Stimulus

Integrating stimulus mechanisms across multiple languages is required in order to provide a full featured UVM-ML environment. Using the type mapping and data serialization concepts described earlier allows for the transfer of

sequence items as well as sequences between languages. UVM-ML provides a set of TLM ports and implementations to facilitate the passing of data between the sequencers of different languages. These ports help to define a "proxy" sequencer that can be used in the foreign language and interact with the native language sequencer of the UVM component being used to drive stimulus

UVM-ML defines mechanisms for splitting the randomization between the different languages such that some fields are randomized first by the test and then later additional fields are randomized in the native language of the component. SDL has found that for the use cases encountered keeping the randomization entirely within a single language greatly reduces the complexity and keeps the interface consistent from the test writer's perspective. The primary downside to doing the randomization this way is that it requires providing (or duplicating) all of the necessary constraints into both languages. This adds some overhead and maintenance to the solution, but it is a tradeoff between support costs on a smaller component development team vs sequence test complexity which would be seen by a much larger set of DV engineers.

In sophisticated UVM components, it is not uncommon for the sequencer to provide functionality to its sequences to aid in creating interesting and complex stimulus. With UVM-ML, the test writer no longer has direct access to the native sequencer that the sequences will be run on. It takes some thought and care to develop an API that will be exported to the foreign language for use by the remote sequences. One example of this from the use cases is that the packet contains an extensive amount of code that was needed in `post_randomize()` to calculate some of the fields that could not be randomized. To solve this, a non-blocking transport TLM port was added to "finalize" the packet. The transport TLM allows the packet to be sent across to the native language, finalize the packet as if `post_randomize()` had been called, and then update the contents back into the original packet so that from the user's perspective `post_randomize()` just gets called as expected even though it was in a foreign language. Using the TLM ports in this way greatly reduce the amount of code that would need to be present in both languages, allowing for effective reuse of the content.

Example 1: Driver / Packet post_randomize() API

```
extend abc_seq_driver_proxy {
  !finalize_pkt_out: out interface_port of tlm_nonblocking_transport of (abc_pkt, abc_pkt);
};
extend abc_pkt {
  post_generate() is also {
    finalize_pkt();
  };
  finalize_pkt() is {
    var tmp: abc_pkt;
    if(finalize_pkt_out == NULL) {
      if(driver != NULL) {
        finalize_pkt_out = ml_driver.finalize_pkt_out;
      } else {
        dut_error("abc_pkt finalize_pkt can't find valid TLM port");
      };
    };
    compute finalize_pkt_out$.nb_transport(me, tmp);
    // Copy back all fields from tmp to me that can be changed by finalize_pkt
    // Details up to implementation (could use reflection, or just plain asignments)
  };
};
```

```
class abc_ml_proxy_env extends uvm_env;
  `uvm_nonblocking_transport_imp_decl(_finalize)
  uvm_nonblocking_transport_imp_finalize#(.REQ(abc_pkt), .RSP(abc_pkt),
    .IMP(abc_ml_proxy_env)) finalize_pkt;
  function bit nb_transport_finalize abc_pkt req, output abc_pkt rsp);
    req.finalize_pkt();
    rsp = req;
  endfunction
endclass
```

While UVM-ML provides the basics for supporting stimulus across the language, one area that it did not natively support was the handling of deferred responses. Many protocols support a deferred response capability where many requests may be issued at a time and responses come back much later in time, potentially out of order with respect to the original requests. UVM-SV supports this in sequences through the "`get_response()`" API. In order to proxy that same functionality across the UVM-ML framework, SDL extended the existing UVM-ML proxy sequencers to support the feature. This was done by wrapping all `sequence_items` in a "tracker" structure that contains a globally unique identifier. For Specman/e originating transactions, this identifier is provided by the Specman/e "`sn_unique_id_for_struct`" function. Both sequencers then use this tracker to keep track of the requests that are active (based on the specific requirements of the protocol). When the UVM-SV side is tracking the request it spawns a new task to monitor the deferred response using the sequence "`get_base_response`" API. Once the response is available it uses the tracker `struct` to send the response through a non-blocking put port though UVM-ML. By using the unique identifier the original sequencer is able to find the original request and trigger the equivalent "`response done`" handling as was provided in the UVM-SV component. The code provided below in Example 2: Deferred Response Handling provides the majority of the code used to implement the deferred response mechanisms.

Example 2: Deferred Response Handling

```
extend abc_seq_driver_proxy {
  !tracked_reqs: list(key: struct_id) of abc_ml_tracker;

  deferred_response_imp: in interface_port of tlm_nonblocking_put of
    (abc_ml_tracker) using suffix=_deferred_resp is instance;
  keep bind(deferred_response_imp, external);

  get(req : *any_sequence_item)@sys.any is {
    var pkt: any_sequence_item;
    pkt = get_next_item();
    // Instead of returning the item directly we wrap it in a "tracker" for
    // later processing the deferred response
    var tracker: abc_ml_tracker = new with {
      .pkt = pkt.as_a(abc_pkt);
      .struct_id = sn_unique_id_for_struct(pkt);
    };
    track_request(tracker);
    req = tracker;
  };
  track_request(tracker: abc_ml_tracker) is {
    // Add qualifications for request tracking so that only packets/requests
    // that eventually receive a response are added to the list
    tracked_reqs.add(tracker);
  };
  try_put_deferred_resp(resp: abc_ml_tracker): bool is {
    if(tracked_reqs.key_exists(resp.struct_id)) {
      var req: abc_pkt;
      req = tracked_reqs.key(resp.struct_id).pkt;
      // Call whatever "response done" processing API is used
      req.set_response_done(resp.pkt);
      tracked_reqs.delete(tracked_reqs.key_index(resp.struct_id));
    } else {
      check NO_TRACKED_REQ_EXISTS that FALSE else
        dut_error("No tracked request exists with struct_id=", resp.struct_id);
    };
  };
};

class abc_ml_seqr_tlm_if extends
  ml_seqr_tlm_if#(abc_pkt,abc_base_sequence);

  uvm_nonblocking_put_port #(abc_ml_tracker) deferred_response_p;
```

```
  virtual task get_item(output uvm_sequence_item req);
    uvm_sequence_item tmp;
    abc_ml_tracker tracker;
    send_item_p.get(tmp); // get item
    if(!$cast(tracker, tmp)) begin
      `uvm_fatal("INCOMPATIBLE_ITEM_TYPE", $sformatf(
         "Expecting abc_ml_tracker from send_item_p.get(), but recieved: %s",
         tmp.get_type_name()));
    end
    req = tracker.pkt;
    current_struct_id = tracker.struct_id;
  endtask : get_item

  virtual task update_response(item_seq s, uvm_sequence_item req, uvm_sequence_item
response);
    abc_pkt ireq;
    if(!$cast(ireq, req)) `uvm_fatal("get_response","Cannot cast response");
    // Associate the req transaction_id with the ML struct_id so the SV response
    // can be associated back to the ML request later
    ml_struct_id[req.get_transaction_id()] = current_struct_id;
    current_struct_id = -1;
    if(ml_parent_seq == null) begin // Only fork once on the first time through here
      // This relies on the fact that the item_seq is reused for every request coming
      // through the seqr_tlm_if
      // See $UVM_ML_HOME/ml/primitives/sequence_layering/sv/seqr_tlm_interface.sv
      // for more details
      ml_parent_seq = s;
      fork
        monitor_deferred_responses();
      join_none
    end
    get_response_p.put(req); // send item_done response
  endtask : update_response

  virtual task monitor_deferred_responses();
    uvm_sequence_item rsp_item;
    abc_pkt deferred_rsp;
    int struct_id, trans_id;
    abc_ml_tracker tracker;
    forever begin
      // Wait for a new response
      ml_parent_seq.get_base_response(rsp_item);
      if(!$cast(deferred_rsp, rsp_item)) begin
        `uvm_fatal("RESP_TYPE_NOT_SUPPORTED", $sformatf(
          "Expecting abc_pkt for response but got %s", rsp_item.get_type_name()));
      end
      trans_id = rsp_item.get_transaction_id();
      // Find the ml_struct_id for the tracked request
      if(ml_struct_id.exists(trans_id)) begin
        struct_id = ml_struct_id[trans_id];
        // Create new tracker to associate this response with the request back
        // on the foreign language ML component
        tracker = abc_ml_tracker::type_id::create("tracker", this);
        tracker.struct_id = struct_id;
        tracker.pkt = deferred_rsp;
        // Put the tracker through the TLM to the foreign ML component
        deferred_response_p.try_put(tracker);
        ml_struct_id.delete(trans_id);
      end else begin
        `uvm_error("NO_ML_MAPING_FOR_RESP", $sformatf(
```

```
          "Could not find ml_struct_id for SV seq transaction_id=%0d", trans_id));
      end
    end
  endtask
endclass
```

One feature common in many of the interface VCs being developed was the need for active agents to act as an "auto-responder" for requests monitored on the interface. This was done in the UVM-SV VC by automatically creating response sequences from a common response sequence base class. Within a purely UVM-SV test bench, it would be possible to change the behavior of the response sequences by using class extensions of the response sequence along with type overrides in the UVM factory. While this would still be possible from a UVM-ML environment, it would have exposed more of the multi-language aspects to the test writers primarily writing Specman/e code. Instead SDL created several configuration modes and structures that are interpreted by the response sequence to change its behavior. The result is a comprehensive set of functionality to support various auto-responder behaviors. The down side is the need to edit a common source for all of the functionality. SDL chose to place a slightly higher burden on the VC development team instead of pushing the extra complexity to the much larger group primarily using UVM-e for test development.

UVM-ML Environment Integration

UVM-ML provides as part of the framework the ability to build a test bench using a unified hierarchy. This allows a UVM test bench in one language to create UVM components in any of the other languages and present them as a single unified UVM hierarchy as if they were all part of the same language. One of the major benefits of this is the ability to make the instantiation of a VC have the look and feel of a native VC even when the actual VC is written in a foreign language.

For all of the use cases involving internally developed VIP, the VCs developed by SDL provide a proxy environment to represent the foreign language VC. This includes providing all of the same layers as a typical UVM VC such as an env, agent, driver, sequencer, and monitors. Some of those layers are not strictly necessary, but providing them helps to insulate the users from the multi-language details.

One area that continues to be a challenge is in the handling of passing virtual interfaces into the test bench for the UVM-SV VCs. UVM-e has a very different mechanism for managing DUT interface signals so the UVM-SV methods are unfamiliar to most of the UVM-e test bench owners. This is an area where it isn't easy to keep the UVM-SV/UVM-ML aspects simplified from the integrators perspective. Training the developers on this integration requirement has mitigated this complexity, but further exploration into ways of simplifying the DUT connections is warranted.

Through the unified UVM hierarchy, UVM-ML provides the functionality of passing configuration information between languages using standard UVM configuration mechanisms. UVM-SV makes use of the standard `uvm_config_db set` and `get` routines whereas UVM-e can use `uvm_config_set` and `uvm_config_get keep` constraints to handle passing configuration data between the languages.

All of the VIPs developed in the use cases take advantage of this multi-language configuration capability. This is done primarily through a configuration object that acts as a container for each VC. Using the configuration object reduces the number of calls and mechanisms needed to pass information and allows for good debug for looking at all configuration of a VC at a glance.

Care must be taken when using the UVM-e '`keep uvm_config_set`' calls as there are specific generation order requirements that must be met. The full details are explained in the UVM-ML reference guide [2]. Unfortunately it is easy for a user to inadvertently create constraints that break the generation order. To combat this problem, SDL added a sanity check to the configuration setup to check for inconsistencies that can result from unsupported constraints. The `config_object` that is passed via `uvm_config_set` from UVM-e is captured locally by making a deep copy and then during `post_generate`, the copy is compared to the original to verify that no new constraints were applied. If any differences are found the tool emits an error informing the user of the difference and provides a reminder of the type of constraint that can lead to the error, see Example 3.

Example 3: Checking Proper Config Struct Generation with keep uvm_config_set

```
extend abc_ml_env_u {
  config: abc_config_obj;
  !ml_check_config: abc_config_obj;
  config_dbg(cfg: any_struct, msg: string): any_struct is {
    message(LOW, "uvm_config_set(", me.e_path(), ".*sv_proxy*,", msg, ")") {
```

```
      print cfg;
    };
    result = cfg;
    if(cfg is a abc_config_obj (my_cfg)) {
      ml_check_config = deep_copy(my_cfg);
    };
  };
  keep uvm_config_set("*sv_proxy*", "config_obj", config_dbg(config, "config_obj"));

  post_generate() is also {
    var cfg_diffs: list of string;
    cfg_diffs = deep_compare(config, ml_check_config, 100);
    if(!cfg_diffs.is_empty()) {
      // Print differences and thrown an error
    };
  };
```

Several of the VCs in the use cases explored required the use of parameters within the UVM-SV component. Typically this has occurred when the SV interface itself is parameterized. In order for the driver, monitor, etc. to get the correct virtual interface, those components must know the parameter values. This requires passing the parameter values down through the UVM-SV component hierarchy. This poses a challenge for the unified hierarchy with UVM-ML as the instantiation of a UVM-SV component from UVM-e is done using a string based lookup in the UVM factory. To solve this problem, the VCs implement a string representation of the set of parameterized VCs that can be created. Those strings are then used to add explicit strings to the UVM factory for each possible use case. See Example 4 for details. This can get quite verbose if there are large numbers of parameters with many valid permutations. This has not been an issue in the cases evaluated here, but some form of automation could be used if the list grew too large or complex.

Example 4: Using Parameterized UVM-SV Components with the Unified UVM-ML Hierarchy

```
//From Specman/e env instantiating an UVM-SV component:
sv_proxy: child_component_proxy is instance;
keep sv_proxy.type_name == appendf("SV:abc_ml_proxy_env(%0d)", config.pkt_width);
// If using any external class based method ports set external_uvm_class
keep EXT_UVM_CLASS is me.external_uvm_class() ==
appendf("abc_ml_pkg::abc_ml_proxy_env#(.PKT_WIDTH(%0d))", config.pkt_width);
```

```
// From UVM-SV package where the component is defined:
  static uvm_pkg::uvm_component_registry #(abc_ml_pkg::abc_ml_proxy_env#(128),
"abc_ml_proxy_env(128)") abc_ml_pkg_forced_registration_type_128;
  static uvm_pkg::uvm_component_registry #(abc_ml_pkg::abc_ml_proxy_env#(256),
"abc_ml_proxy_env(256)") abc_ml_pkg_forced_registration_type_256;
```

B. Advanced UVM-ML Functionality

UVM-ML Register Model Integration

Through many of the features already built-in to UVM-ML, we are able to relatively seamlessly integrate components and write tests using multiple languages in a single environment. However many types of testing and components need to have access to a register model for interacting with a DUT's control registers or other internal memories. Both UVM-SV and UVM-e include register models as part of their methodology, but the two implementations are distinct and separate. UVM-e uses a model called "vr_ad" whereas UVM-SV has the UVM Register Layer implementation. As part of the base UVM-ML OA implementation, there is a TLM interface provided to access a vr_ad based register model from UVM-SV. This provides a basic interfaces for reading, writing, peeking, and poking registers from UVM-SV through the vr_ad model, however it does not include a mode that provides the same seamless type of integration as the stimulus aspects do. To make the model more usable from both languages, SDL implemented a proxy mechanism to bridge between the UVM-SV register layer model and the UVM-e vr_ad model. Our lab uses an internally developed tool for describing registers and generating the vr_ad model. This tool also supports building UVM-SV register models and so we are able to generate equivalent register models in both

languages. Then through the use of the proxy as show in Figure 1, we are able to proxy the register information between languages. This proxy provides a true multi-language environment enabling the UVM-SV and UVM-e components to interact with their language native register models. The proxy is able to keep both register models up to date and consistent.

The implementation consists of a verification component that acts as an active VC that would normally be used to drive the DUT register interface. Instead, it proxies all of those requests through to the vr_ad model in UVM-e. The vr_ad then takes complete responsibility for reads/writes both front-door and backdoor to the DUT registers. Any updates to registers that the vr_ad model observes are also conveyed to the UVM-SV register model as if the update had been monitored directly. This allows tests/sequences to be written in either language while keeping all the register states in sync. Likewise checkers, scoreboard, or other components that need access to the register information can use their native register model calls to access. Through this type of integration, UVM-e and UVM-SV components are not generally aware that the accesses are occurring in another language. This provides for greater reuse of components as a checker written for UVM-SV can be used both in a UVM-ML environment as well as a UVM-SV only environment with no changes to the code.

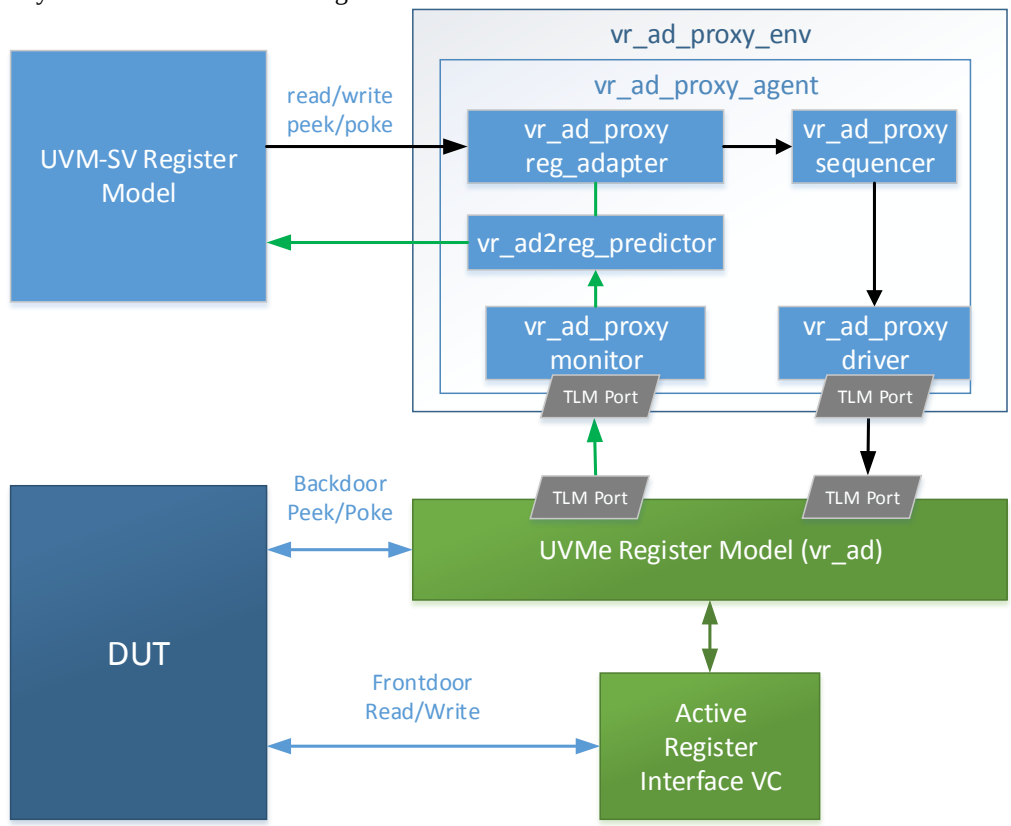

Figure 1: UVM-ML vr_ad_proxy

UVM-ML Runtime Phase Synchronization

As of version 1.8 of the UVM-ML OA, runtime phase synchronization was not supported. As of 1.9 this feature is now supported (for Cadence, in Xcelium simulator only). SDL has implemented its own phase synchronization functionality which should be simulator agnostic. Our implementation is mastered from the UVM-e side and uses a polling loop to periodically check if the UVM-SV side is still busy. At the start of a UVM-SV runtime phase, if the phase is setup to sync with an UVM-e testflow phase, the synchronizer immediately raises an objection on its own to prevent the phase from advancing prematurely. The synchronizer does not drop this objection until instructed to do so from the UVM-e side of the synchronizer. At each iteration of the polling loop in UVM-e, it queries if the UVM-SV synchronizer is still busy. This busy function returns `true` if the phase is being synced and the objection count is greater than 1. Note, it checks if the objection count is greater than 1 (not 0) as there will always be 1 objection from the synchronizer itself. The busy must also return `true` during the transition of the phases. When the synchronizer is told to advance phase, it drops its objection and sets a flag to indicate that it is still advancing. That flag causes the

synchronizer to report busy until the next UVM-SV runtime phase starts. This prevents race conditions between UVM-e advancing phase and UVM-SV advancing phase.

For projects using the runtime phases, providing this sort of synchronization is a valuable feature as it allows to the tests, sequences, and other aspects of the verification environment to stay in sync across languages. Again this type of functionality serves to make the fact that another language is even involved in the environment transparent. Components and sequences can raise objections in either language like would typically be done and the two languages just stay in sync.

III. UVM-ML Case Study

A. Porting UVM-e VCs to UVM-SV/UVM-ML

This paper discusses two interface VCs that were converted from UVM-e to UVM-SV including support for UVM-ML. These two VCs were ported because the usage of them needed to shift from internal use only VIP to ones that would be shared broadly across many companies. While UVM-e offers many advantages to DV teams, it is clear that many companies have adopted UVM-SV and components in UVM-SV are expected by companies looking for 3rd party VIP solutions. UVM-ML made it possible to transition these interface VCs from UVM-e to UVM-SV with a minimum of disruption to existing test benches and tests across four different IP block environments and three chip environments.

The two VCs that were ported are the Gen-Z [3] VC and CODI Tier1 VC. The CODI Tier1 is a typical interface VCs that implement the wire protocol for interfacing with a DUT. CODI Tier1 is an HPE developed on die interconnect used by many internal projects. The Gen-Z VC is an abstract model for representing Gen-Z packets and protocol. The detailed workings of these VCs is beyond the scope of this paper, but the figures below are included to help provide context to the reader as to the complexity of the problem and solution of this use case. Figure 2 provides diagrams that show the layering aspects of the Gen-Z VC along with the CODI Tier1 and Gen-Z PLA VCs. Figure 3 provides a high level overview of how the UVM-ML aspects interact with the common Gen-Z VC.

The Gen-Z VC is layered on top of multiple physical interfaces including the CODI Tier1, but also includes other interfaces such as the Gen-Z Physical Layer Abstraction (PLA) [3] interface. This layering is done to keep the higher level protocol handling common and allow for reuse of sequences across different types of chip interfaces. The primary UVM-ML component was developed to interface with the Gen-Z VC layer of these use cases. It consists of proxy sequencer and monitors to provide the same look and feel as other UVM-e components. The UVM-e monitors enabled connection to all of the other components in the system such as checkers, scoreboards, as well as chip protocol performance analysis tools. Through many of the mechanisms described previously, the existing test benches and tests were able to continue running largely unchanged.

The Gen-Z VC was directly ported, nearly line by line, from UVM-e to UVM-SV. In most cases the UVM-e/aspect oriented functionality was able to be refactored to fit within the limits of SystemVerilog. In most cases, this simply meant combining multiple aspects of a function from e ("`is also`" extensions) into a single SV function using `if` statements or `case` statements to take the different flows. There were also cases where code had to be changed more dramatically. One such area related to the tag management functionality being used by one of the block teams. The team had made significant customizations to fit their needs through subtypes declared outside of the primary component code base. Since the underlying tag management function moved to UVM-SV from UVM-e, that mechanism was no longer available. Through this transition, we developed a new portion of the VC that acts as a "tag manager". The block team could then configure and/or extend that tag manager using UVM-SV mechanics to customize the functionality needed by that team.

Through this conversion, we were able to compare the amount of code needed to implement the same component in two different languages. The UVM-e implementation of the Gen-Z VC was approximately 4000 lines while the UVM-SV content was 4600 lines. The UVM-ML (e and SV) content accounted for about 600 additional lines. Through this conversion, we grew the total lines of code by about 25% over all, but half of that growth was in the UVM-ML content itself.

The CODI Tier1 implementation made extensive use of aspect oriented features from e and was not able to be ported line by line in the same way as the Gen-Z VC. Many of the algorithms were able to be leveraged from the aspect oriented approach but had to be significantly refactored to fit the SystemVerilog language. Within HPE CODI is a tiered interface and there are numerous implementations that reuse a common Tier1 (wire layer) but have separate packet protocol layers (Tier2+). The upper tiers are mostly aspect oriented extensions of the Tier1 implementation and makes use of the ability to mix multiple "`when subtypes`" in a single component all working in conjunction. The single inheritance object oriented mechanisms in SystemVerilog could not easily accommodate the original code structure. It would have been possible to collapse the code down using a lot of conditional code in each of its processing steps, however we did not pursue this option as the Gen-Z use case was very limited in the number of

features needed. Instead of supporting every CODI feature, it was decided to develop a simpler version that was targeted to the required feature set rather than the larger CODI ecosystem feature set within HPE.

Part of the motivation for using UVM-ML was to be able to reuse the existing tests and environments. With these implementations, we were able to keep all of the same test environments with minimal disruption. The stimulus content was almost entirely reused with extremely minimal changes needed. Specman/e provides many advantages for sequence/test development beyond UVM-SV. Aspect oriented programming provides for a very easy method for adding or changing constraints in sequences and test bench configuration. While many of the same functions can be replicated in UVM-SV, the amount of code needed is often greater. UVM-SV is also limited because of its single inheritance object model. Sequence "aspects" can be defined as multiple "`when subtypes`" each contributing different constraints or functions. Those various aspects can then be mixed in numerous cross products of settings. This would be difficult in UVM-SV unless using a very flat structure. Another feature for Specman/e is the ability to define "constraints from above." This is a feature unique to Specman/e that allows a higher layer sequence to dynamically "attach" constraints to an item that will be randomized multiple times. All of these features of Specman/e have been advantageous to us in developing our testing and we believe the benefits have been worth the costs.

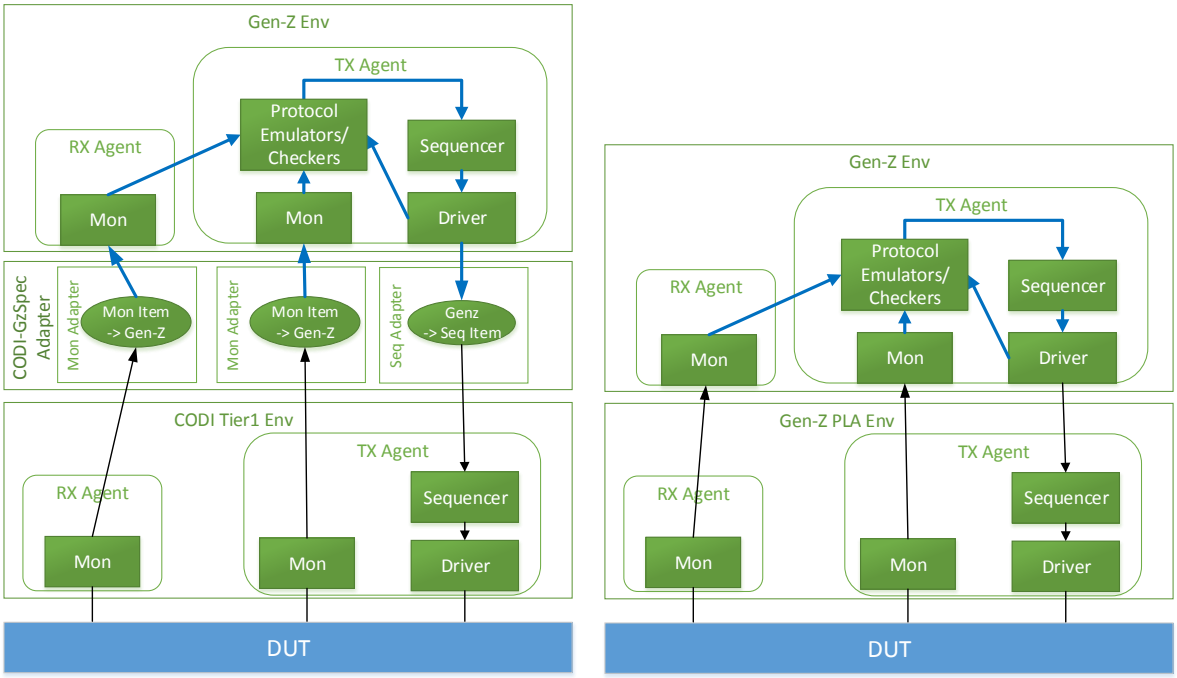

Figure 2: Gen-Z CODI and Gen-Z PLA Interface VCs

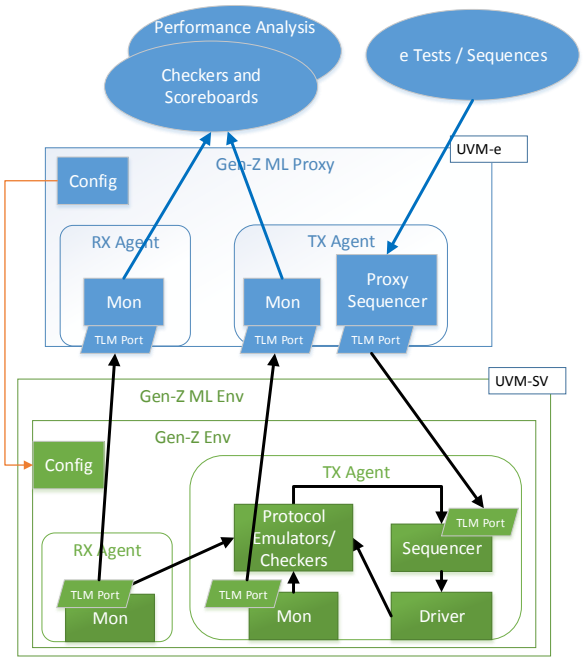

Figure 3: Gen-Z Interface VC Multi-Language Overview

IV. Conclusions

SDL has been using UVM-ML functionality for multiple years across numerous projects and designs. The use cases discussed in this paper have been in use for over a year and a half. Through these projects, SDL has been able to take advantage of many of the advanced testing features available in Specman/e while utilizing a variety of UVM-SV content from internally developed VCs to externally purchased Verification IPs. This technology has also enabled us to make content available in a language native way across both UVM-e and UVM-SV test benches. There certainly have been costs to developing the content to be multi-language aware, but that scope has been fairly localized to a small component development team. Through the capabilities already present in UVM-ML as well as those added by SDL itself, the larger integration and test teams have been insulated from most of the multi-language aspects.

So is Specman still relevant? We believe yes! It is still relevant to our industry as it has many advanced features that are not available in other languages. However UVM-ML is a necessary component of that solution as it is clear that no one language has yet filled all of the needs of the verification industry.

V. References

[1] "UVM (Universal Verification Methodology)," [Online]. Available: http://accellera.org/downloads/standards/uvm.

[2] Accellera, "UVM-ML Users Guide," [Online]. Available: http://forums.accellera.org/files/file/65-uvm-ml-open-architecture/.

[3] "GenZ Consortium," [Online]. Available: http://genzconsortium.org/.