Multi-Language Verification: Solutions for Real World Problems

Bryan Sniderman
Advanced Micro Devices, Inc.
Verification Methodology
Toronto, Canada
bryan.sniderman@amd.com

Vitaly Yankelevich
Cadence Design
Advanced Verification Solutions Division
Rosh Ha'Ain, Israel
vitaly@cadence.com

***Abstract*—*Modern verification projects often need to deal with a mixture of available off-the-shelf verification components. This presents several challenges. The components may be implemented in different languages (SystemVerilog, e, SystemC, C++) based on different methodologies (flavors of UVM, VMM, different C++ class libraries). The involved actors (integrators and verification IP developers) need to combine these components together with minimal changes. This paper presents use-cases that leverage an open multi-language architecture addressing those challenges. The paper provides an overview of the solution and illustrates how it can be applied to one of the emerging use cases.*

Keywords— functional verification; verification intellectual property (VIP); verification reuse; unified verification methodology (UVM); multi-language; transaction-level modeling (TLM) ; backplane

I. INTRODUCTION

Subsystem and system-level verification face many challenges on different scales: modeling and refinement at different levels of abstraction, verifying proper functionality and performance of complex systems, integration checking and so on. The need to deal with several implementation languages and diverse verification methodologies is only one of those challenges.

This paper begins with a description of the relevant use cases requiring a multi-language solution. Following that, it presents the architected solution (UVM-ML OA) that enables efficient integration of multi-language verification IP (VIP) components. UVM ML OA helps to reduce or eliminate the time and effort that verification architects need to spend on dealing with multi-language integration related issues. This paper then describes a simplified hypothetical subsystem-level use case that illustrates the usage of the UVM-ML OA mechanisms for different tasks.

A. VIP Reuse (Vertical and Horizontal) Use Case

Time to market and development efficiency are important drivers for reuse in general. Applied to functional verification, vertical and horizontal reuse reduces duplication of efforts in developing testbenches and test scenarios at the IP and system on a chip (SoC) levels [1],[2].

The need to reuse even one useful VIP, implemented in a different language, is a sufficient reason to necessitate a multi-language environment. IP, subsystem and system-level testbenches which employ a re-use methodology, often bring together multiple verification IPs (VIP's), one or more reference models, scoreboards and virtual sequences that may be written in different languages. In such a system, the configuration settings, data transactions, sequences, stimuli, and synchronization coordination may cross the language boundaries.

Figure 1 illustrates the typical reuse of an IP level verification environment on a subsystem level.

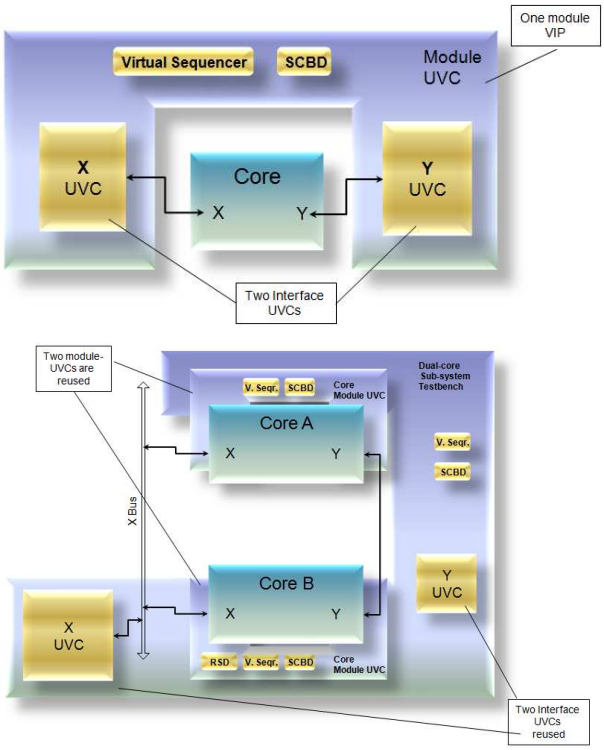

Fig. 1. Vertical reuse: IP level vericiation environment and its integration in the subsystem level.

B. *Transformation of a Design Under Test Through Multiple Abstraction Levels Use Case*

[3] and [4] describe the real-world use cases where the design under test (DUT) is modeled in different languages at different abstraction levels.

As the DUT abstractions evolve, the testbench should evolve in parallel to incorporate the different levels of abstractions into it. For example, as shown in Figure 2, a DUT can be initially represented by a high level SystemC functional model. This SystemC model enables development of the testbench ahead of an RTL model and can also be embedded into a scoreboard.

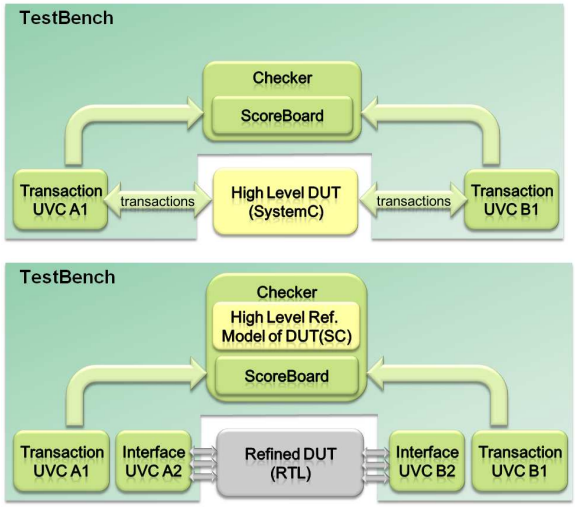

Fig. 2. Testbench reuse with different design-under-test models.

This naturally leads the testbench to evolve into multi-language. Leveraging the standard transaction-level modeling (TLM) in the testbench aids in making it invariant to the abstractions and to the implementation languages of the DUT model. A testbench architected with these enabling facilities is able to support different abstractions or even concurrent mixed-abstractions seamlessly.

C. *Hardware-Assisted Verification With an Accelerated VIP Use Case*

[5] and [6] present the use cases where the accelerated VIP helps reduce verification time. They make the premises that the testbench won't require significant reworking or present debugging challenges when switching between software-based simulation and hardware-based acceleration environments.

This use-case can be viewed as a specialization of the previously referenced multiple abstractions DUT use-case.

Figure 3 shows a synthesizable version of the DUT placed in an acceleration device (such as FPGA, or HW-accelerator). The bus functional model (BFM) and passive collector components of the testbench are also included in the synthesizable part of the design. A specialized software transaction layer (implemented, for example in C++) abstracts between the simulated and synthesized domains.

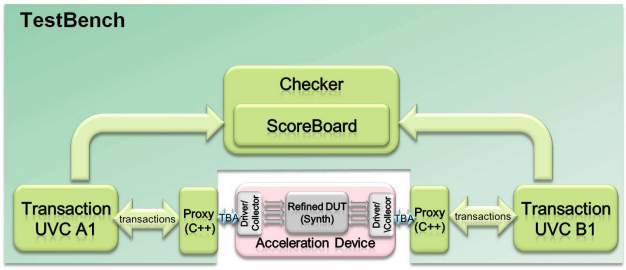

Fig. 3. Accelerated verification IP.

Both the software layer and synthesizable part of the testbench would benefit from leveraging standardized multi-language TLM communication, phasing, and hierarchical configuration facilities which are portable to different domains.

D. *Software-Driven Functional Verification Use Case*

Software-driven verification has been gaining momentum over the past few years. [7] and [8] present different mechanisms for using the software to control test environment. [7] presents the usage of TLM2.0 virtual platforms and [8] describes how SW tests can control a VIP via the virtual register interfaces as shown on Figure 4.

The main goal is to enable real world software scenarios to be executed on a DUT (IP or subsystem) in order to exercise it in the way it is intended to be used. Conversely, an abstraction of a DUT can be provided to the associated software teams in order to enable early software development. In both cases, a software environment (e.g. a virtual machine) is manifested in one language, and a simulation or acceleration environment is deployed in another. Regardless of the specific methodology, leveraging a seamless multi-language platform for interoperating between domains is necessary in order to enable rapid integration.

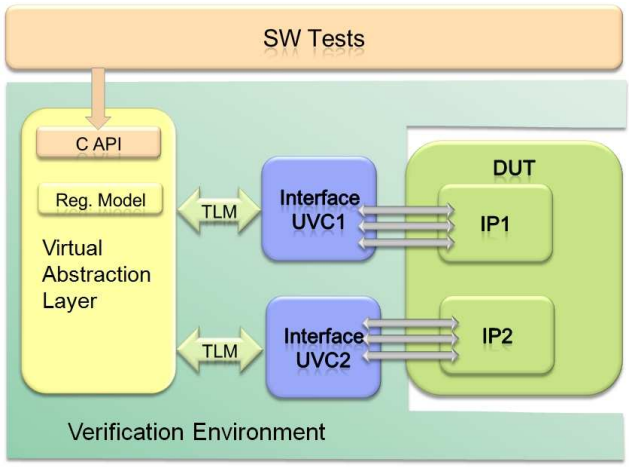

Fig. 4. Software-driven functional verification

II. Previous Work

There is a long list of former and currently-available solutions enabling mixed-language communication. They can

be grouped into 2 categories: multi-language co-simulation solutions and multi-language verification methodology solutions.

The first category, which includes EDA vendor simulators and co-simulation backplanes, targets low-level integration of hardware-description languages (HDL) and simulation engines. This level is targeted primarily for design purposes and it deals with issues such as signal propagation, fine-grain synchronization of simulation engines, and so on. Another characteristic of the low-level communication is that it does not support an object-oriented programming paradigm. In particular, it does not address class-based communication between the languages. Based on these criteria, we can include in category the C-language programming interfaces, such as the standard SystemVerilog DPI, the Specman C interface, and similar C-based interfaces.

The second category aims at enabling integration of high-level verification methodologies, such as SystemC TLM, UVM SystemVerilog, UVM-e, OVM, and VMM, to name a few. The focus of this category is to align the main common methodological constructs, such as testbench phasing, configuration, transaction-level communication, and test selection. This category requires support of the class-based interfaces. This multi-language solution described in this paper falls under the second category.

Products already available on the market provide partial solutions for this area. Each of these products however has some limitations.

For example, Cadence Design developed a UVM multi-language library that allows UVM SystemVerilog, *e* and SystemC to be interconnected via TLM ports. In addition to the multi-language communication, Cadence's solution includes a UVM-based SystemC class library, which can be connected to the other languages. This product is proprietary to Cadence and it does not allow the users to incorporate additional languages or methodologies.

Synopsys VCS-TLI supports transaction-level communication between SystemVerilog and SystemC [9]. It is limited by being non-portable between the simulators and integrates only those two frameworks.

Mentor Graphics' open-source UVM Connect (UVMC) library [10] allows connecting UVM SystemVerilog and SystemC via TLM ports. It also provides a procedural interface for accessing other UVM SystemVerilog facilities from SystemC. This library is portable and runs on different simulators. It was designed to only support point-to-point integration between the two languages, and requires that UVM SystemVerilog is put in charge of providing the methodology-level services to SystemC. As such, all the methodology facilities in the SystemC framework cannot be used independently and require presence of UVM SystemVerilog.

The Accellera Multi-Language Work Group (MLWG) was publicly established in April 2013 [11]. It established the key requirements that a complete multi-language solution should be portable between simulators, should be extensible to additional languages and methodologies, should not depend on any specific methodology library, and should support a broad spectrum of high level services. None of the above-mentioned solutions currently address all of these requirements.

III. UVM-ML Open Architecture

As discussed in the introduction, there is a broad and growing need for integration of the multi-language verification components in modern verification environments. The contents of the aforementioned papers (and many other papers published) reveal that a great variety of the languages and methodologies are used in practice: UVM SystemVerilog, SystemC, e, VMM, OVM, C++, Python and more. This indicates there is strong industry need for a generic solution that addresses this problem and results in saving the engineers from having to devise a proprietary solution in each project.

This section describes the purpose and design of the open source package named UVM-ML Open Architecture (UVM-ML OA), jointly developed by AMD and Cadence. It aims at enabling rapid integration and re-use of multi-language verification components. The term *framework* is used to denote an assembly of verification and modeling facilities implemented in a single language. Frameworks may be written in a specialized Hardware Verification Language (HVL) such as SystemVerilog, or *e*, a modeling language (such as SystemC), or a generic programming language (for example, C++). Examples of frameworks include UVM SystemVerilog, UVM *e*, ASI SystemC, VMM. Different frameworks can be deployed in the same language (for example, UVM and VMM are both in SystemVerilog). A new framework can be composed from a few simpler frameworks in the same language (for example, a combination of ASI SystemC with a UVM SystemC library). The term multi-language used throughout this paper is extended to include the diversity of *frameworks* being integrated, regardless of their target languages.

UVM-ML Open Architecture is so named because it follows the primary concepts defined by the UVM methodology. The qualifier "Open Architecture" emphasizes the intended openness of the solution to enable integration of multiple frameworks, rather than limiting the solution to the fixed selection of languages.

To further clarify the scope of the work, we need to emphasize that UVM-ML OA focuses on bridging the languages and frameworks on the methodology level, using the currently available language constructs. The authors did not attempt to address the mixed language challenges with invention of new specialized extensions, such as, for example, supporting passing objects directly by reference, or calling "foreign" class methods in few selected languages.

The primary goal is to isolate the concerns of the VIP developer and integrator. The developer should not be concerned about whether the VIP will be used in a single-language or a multi-language environment, so long as the developer is following the recommended UVM methodology practices. The integrator is expected to have exposure to the nature of the environment (single- or multi-language). By leveraging the multi-language aware facilities which enable the integrator, for example, to bind TLM ports across the language boundary or to instantiate a "foreign" framework VIP

hierarchically, the integrator can assemble multi-language components and environments.

The UVM-ML OA software distribution package [12], [13] comprises the backplane, reference frameworks and multi-language (ML) adapters. All the open source components of the package are licensed under the Apache License.

A. Backplane

The core of the package is a ***backplane*** shared library. The backplane serves as a routing layer between two or more integrated ***frameworks*** and holds information about the overall topology, which is necessary for routing. This architecture enables collaboration between the frameworks while abstracting away from specific methodologies and languages.

Figure 5 illustrates the UVM_ML topology where the framework clients are connected to the backplane server, located in the middle of the "star system". Any number of frameworks can be interconnected at the same time.

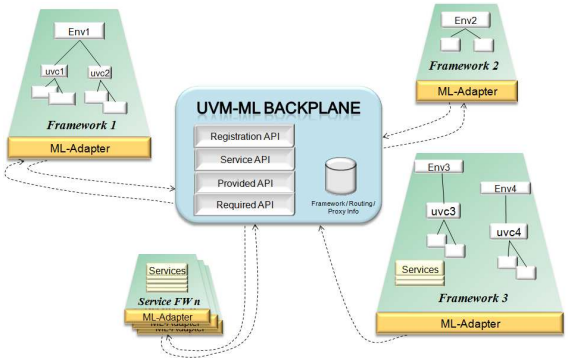

Fig. 5. The backplane and the frameworks.

Communication in the system generally falls into one of the following categories:

- Providing information to the backplane global data repository (for example, the frameworks shall register themselves upon initialization)
- Broadcasting messages from a service provider to the rest of the frameworks (for example, the phasing service is implemented in this way)
- Broadcasting messages from any framework to the rest of the frameworks (for example, the distributed configuration and resources settings)
- Point-to-point communication (for example, passing transactions via TLM)

The content of the messages, flowing between the frameworks, is transparent to the backplane.

The backplane, provided in the UVM-ML OA distribution, is general in nature, and does not need any intervention by users. For example, an end user integrating VIPs into a testbench (an ML integrator) can link in the available backplane library and supported frameworks as is and does not need to be aware of the backplane presence.

B. Reference Frameworks

The frameworks supported in the current release of the UVM-ML OA package include three UVM flavors: UVM SystemVerilog, UVM SystemC, and UVM *e*. In particular, the package provides the patch source code for UVM SystemVerilog with ML enablers, and the source code for the UVM-SC library.

A detailed description of the UVM-SC library is beyond the scope of this paper but we introduce it here briefly to raise awareness of the availability of this framework with the package and of its importance for successful integration of SystemC in a complex ML environment.

UVM-SC is a standalone C++ class library which implements the standard UVM methodology as a framework on top of a standard Accellera Systems Initiative (ASI) SystemC version or a vendor-proprietary SystemC implementation. This framework enables usage of SystemC for high-level verification. The UVM-SC framework capitalizes on the native SystemC facilities, such as threading, SystemC phasing, TLM, events, barriers, and derives from the SystemC base classes as a foundation for the UVM classes. Additionally, UVM-SC builds upon SystemC by leveraging the power of the UVM methodology: standardized testbench components, configuration facilities, common phases, callbacks, resource pools, synchronization extensions, and factory overrides. There are still a few outstanding facilities that should be added to UVM-SC as it matures.

The UVM-SC library was architected from the ground up to be usable in both standalone framework mode (i.e. just UVM-SC) or in a multi-language environment, in conjunction with its adapter. All of the facilities deployed in UVM-SC are scalable to a multi-language environment transparently.

C. Multi-Language Adapters

The previous figure shows the frameworks connected to the backplane via specialized *ML adapters*. The role of the adapter is to connect existing frameworks to the predefined backplane API. The ML adapter provides an abstraction layer between its associated framework and the backplane in order to translate between the two in a seamless manner. Some frameworks may, potentially, connect to the backplane directly, without a special adapter, if they are designed to natively allow redirection of their facilities to an external backplane like server.

D. Services and Facilities

The backplane currently supports the following ML facilities:

- Initialization and registration of the unlimited number of frameworks
- Synchronized phased pre-run, runtime, and post-run execution
- TLM communication
- Hierarchical construction of a multi-language verification environment (a.k.a. unified hierarchy)
- Build-time and run-time configuration

- Resource sharing
- Runtime synchronization between the master and slave frameworks

Some of the deployed facilities require a centralized service provider. This is true for the currently available services, such as phasing, and for the some future facilities that are planned to be added (e.g. messaging, shutdown, and quantum time keeping). The UVM-ML OA backplane is flexible to allow any framework to register itself as a service provider for a specific service. The ML integrator has the option to choose the provider among multiple candidates although a default provider per service always exists.

E. The Challenge of Enabling Multi-Language Facilities

Many frameworks were not designed with the intention of participating in a multi-language environment. Some of the desired multi-language facilities do not easily align between the various frameworks as they are currently written. There may be significant value in synchronizing frameworks better and enhancing them to be multi-language capable, but it will require working closely with the framework developers to extend their frameworks to be ML friendly.

Ideally, the ML features should not require modifications to the existing frameworks however in reality some modifications are necessary. This creates the dilemma of choosing between sacrificing some important multi-language features or trying to add some hooks to enable the features. The features that we could not support with the standard unmodified frameworks implementations were:

- Propagation of the phases (and especially, the "build" phase) between the frameworks, which is necessary to enable unified multi-language hierarchy of verification components
- The ability to configure VIP build-time properties in the process of hierarchical construction (for example, number of agent instances, active or passive agent mode etc.) natively from verification code written in another framework.
- Graceful termination of the test through synchronization of the post-run phases for the involved frameworks
- Runtime checking of TLM connections, with the assumption that all the aligned frameworks finish their build phase before the first framework's connect phase begins. Runtime checking allows issuing connectivity error messages on the spot, with a proper source reference. Catching the error immediately allows for a better debugging experience, rather than deferring the checking until some future time.

Let us take a closer look at phase propagation. Ideally, native test phases of different frameworks should be aligned to some common boundaries (see, for example, this requirement in [14]).

SystemC and UVM SystemVerilog standards define similar phases, but they do not enable alignment between them because the standardized implementations were developed with a single-language use-case in mind and do not provide a public interface for controlling the phasing process from an external controller.

A partial solution, applicable to SystemC, is enabled with introduction of the UVM-compatible SystemC class library (UVM-SC). A structural base class in UVM-SC (*uvm_component*, as in UVM SystemVerilog) is derived from the base class *sc_module* but defines UVM phase callbacks, in addition to the standard callbacks of *sc_module*. If the SystemC testbench developer uses *uvm_component*'s rather than base *sc_modules*, then the SystemC adapter can traverse that hierarchical tree and invoke the UVM-compatible phase callbacks. UVM-SC ML adapter also supports connecting to SystemC TLM ports located in standard SystemC modules, and not only in *uvm_component*'s. In this way it enables seamless ML TLM communication between the SystemC design and testbench domains.

The main disadvantage of this partial solution is that it requires mandatory usage of UVM-SC classes for full phase synchronization. Although the native SystemC phases are supported for *sc_module*'s, aligning the native phases transparently requires an enhancement to the SystemC framework.

A more comprehensive alignment of the phases is achieved by applying some unobtrusive patches to the standard frameworks. The alignment can be coarse-grained or fine grained, depending on the framework or its adapter.

The coarse-grained alignment scheme means that phase alignment only occurs at the boundary of a group of phases. For example, it is rather important that all involved frameworks complete their pre-run phases before execution of the runtime processes or threads, so a coarse-grained deployment would ensure all frameworks align at the end of the pre-run phases before proceeding to the runtime phases. Similarly, all the runtime phases should be finished before any framework starts execution of the post-run phases. Alignment within a coarse-grained group between frameworks would not be supported.

Fine-grained alignment of the phases means that each component's callbacks in the multi-language system are aligned per a common phase, regardless of the framework origin. The fine-grained alignment of the phases is a preferable option, if the framework adapter can support it. Such alignment enables better synergy and makes the whole environment virtually behave as though it is implemented in one language. This results in a more natural approach for integrating multi-language VIP and eliminates any potential side effects that may be incurred with the coarse-grained scheme.

In the UVM-ML OA package, we included patches for the ASI SystemC and UVM SystemVerilog frameworks enabling the fine-grained alignment of the ML phases. The patches add to those frameworks public methods that allow invocation of the phases individually from an external phase controller at the beginning of simulation. These patches do not compromise

backwards compatible behavior and are transparent to users of those frameworks.

The following figure illustrates an example of how the finely aligned pre-run phases for 3 frameworks interoperate: a patched UVM SystemVerilog, a patched SystemC and e.

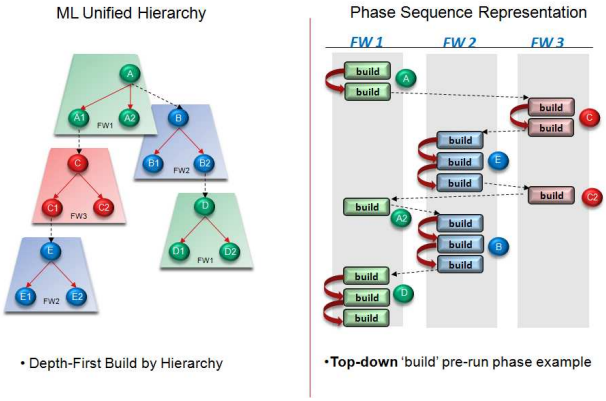

Fig. 6. Time sequence for fine-grained phase aligned frameworks and its corresponding unified hierarchy.

F. Support for ML Configuration and Resources

UVM configuration is a method of sharing the configuration elements via the configuration database. The configuration element comprises a name, value and properties. One property is a context of an originating component.

During the UVM build phase, the context automatically affects the priority property. This allows the highest ancestral parent component to override the values specified by other components on lower hierarchical levels. To emphasize this effect of the hierarchical position of the originating component, we refer to the build phase time configuration as *hierarchical configuration. C*onfiguration during the rest of the phases is referred as *runtime configuration.*

Deploying a multi-language hierarchy requires introducing a multi-language aware configuration facility.

Resources differ from configuration in that the hierarchical context of the originating component is not stored and not used. The remaining characteristics for resources and the configuration elements are common. This allows the ML adapters to maintain both configuration and resource elements in a unified manner.

There are two potential storage models for the multi-language configuration database: the centralized model and the distributed model. Both approaches have their advantages and disadvantages. Developers of the UVM-ML OA have chosen the distributed model based on performance and flexibility considerations. The distributed model enables better performance characteristics because reading the distributed database does not require crossing the language boundaries. The distributed model also allows different frameworks to maintain different (potentially, legacy driven) semantics of interpreting the configuration properties, thus achieving greater flexibility.

The UVM-ML OA backplane API allows a framework to broadcast configuration and resource settings so that they can be stored in the corresponding databases of other connected frameworks. The actual implementation of the user-view and the broadcasting implementation are framework-specific. The reference framework adapters, that are currently included in the UVM-ML OA distribution, support configuration and resource value types that can be passed by copy (singular integral types, strings and serializable objects).

IV. An Illustrative Use Case

This section describes an emerging use case (following the lines of the work presented in [8]), that we use to demonstrate the UVM-ML OA facilities in greater detail. More information about each construct explained below can be found in the open source package documentation [15],[16].

The selected use case represents a verification environment for a hypothetical communication subsystem. A top-level SystemVerilog DUT testbench module *comm_subsystem_tb*, instantiates an abstract module *comm_subsystem_top* and its two interfaces: *host_if* and *comm_if* as shown in *Figure 7.*

```
// SystemVerilog testbench and DUT

import uvm_pkg::*;

interface comm_if(input logic clk);
  wire data [31:0];
  wire addr [31:0];
  string this_name = $sformatf("%m");
endinterface

interface host_if();
  string this_name = $sformatf("%m"); …
endinterface

module comm_subsystem_top (comm_if cif, host_if
hif);
endmodule

module comm_subsystem_tb;
  logic clk = 0;
  comm_if cif (clk);
  host_if hif();
  comm_subsystem_top comm(.cif (cif), .hif(hif));
  …
endmodule
```

Fig. 7. An abstract SystemVerilog design under test.

The verification environment for this testbench is composed of three layers as shown in Figure 8.

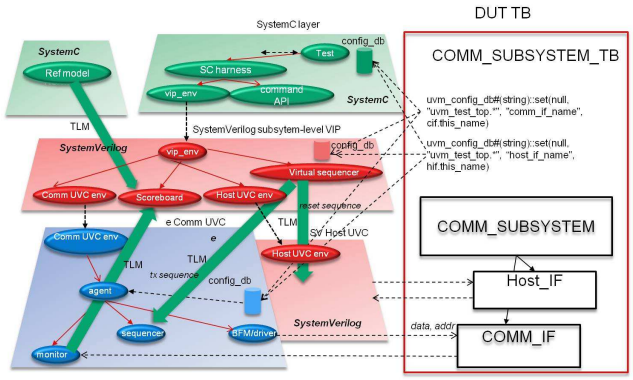

Fig. 8. ML verification environment for an abstract communication subsystem.

The high-level SystemC verification harness layer enables integration with the application software domain. The harness layer is leveraging the UVM-SC framework and its primary task is to mediate between the control knobs, belonging to the software, and a SystemVerilog reusable VIP. Additionally, the SystemC layer includes a reference model. This reference model was originally used as a high-level abstract model of the DUT and is now connected to the scoreboard. While the actual software is not available in this example, the tests provided (see *test1* on Figure 8) are also implemented in SystemC/C++, mimicking the software control knobs. The control knobs in the SystemC are modeled using a command API class. The "toy" command API in this use case represents a C structure, containing a command *op* and few arguments *op1*, *op2* … In a more realistic example; this structure could be represented by a group of registers or a high-level programming library.

```
// SystemC high-level harness environment
#include "tlm.h"
#include "uvm.h"    // UVM-SC topmost header file
#include "uvm_ml.h" // UVM-ML adapter header file
using namespace uvm;
using namespace uvm_ml;

class command_api{// Mimicking SW control knobs
public:
  int      command;  int  op1;  int  op2;
  sc_event command_event;
  …
  void set(int c, int i1, int i2) {
    command = c;  op1 = i1; op2 = i2;
    command_event.notify();
  }
};
class sc_harness : public uvm_component {
public:
  command_api ca; // command API instantiated
  sc_tb (sc_module_name nm):uvm_component(nm), …
  { … }
  void build_phase(uvm_phase *phase) {  … }
  void connect_phase(uvm_phase *phase) { … }
  void run_phase(uvm_phase *phase) { … }
  UVM_COMPONENT_UTILS(sc_tb)
};
class test1 : public uvm_component { // A test
public:
  sc_harness * sc_h; // SC harness component
  sc_test1(sc_module_name nm): uvm_component(nm)
          { sc_h = new sc_harness("sc_h"); }
  void run_phase(uvm_phase *phase) {
    wait(1, SC_NS);
    sc_h->ca.set(RST_SEQ, (-1), (-1));
    wait(10, SC_NS);
    sc_h_>ca.set(TRANS_SEQ, 10,(-1));
    …
  }
  UVM_COMPONENT_UTILS(sc_test1)
};
```

Fig. 9. SystemC verification harness layer.

Figure 9 shows the definitions of the SystemC harness and test classes. Both classes are derived from the base structural class *uvm_component*, similar to its corresponding class in the standard UVM SystemVerilog library. The classes have the same phase callback methods (*build_phase*, *connect_phase* etc.). In this example, *test1* activates the control API in the blocking *run* phase.

Figure 10 demonstrates how the reusable SystemVerilog VIP is instantiated in the SystemC harness layer. It provides a more detailed view of the class *sc_harness* than shown on Figure 9.

```
// SystemC high-level harness class - zoomed in
#include "tlm.h"
#include "uvm.h"    // UVM-SC topmost header file
#include "uvm_ml.h" // UVM-ML adapter header file
using namespace uvm;
using namespace uvm_ml;
…
typedef class build_config_c;// config object
class sc_harness : public uvm_component {
public:
  command_api      ca;
  uvm_component *  vip_env;
  build_config_c * env_config; // Config object

  tlm_analysis_port<uvm_seq_control_base> aport;

  sc_harness(sc_module_name nm):uvm_component(nm)
            ,aport("aport"), vip_env(0)
  { env_config =  new
      build_config_c("SV","comm_vip_env",ACTIVE);
    uvm_ml_register(&aport);
  }
  void build_phase(uvm_phase *phase) {
    vip_env = uvm_ml_create_component(
              env_config->frmw_name,
              env_config->type_name, "env",
              this);
  }
  void connect_phase(uvm_phase *phase) {
    string aexport_name =
      vip_env->name()+string(".")+"control_imp";
    uvm_ml_connect(aport.name(), aexport_name);
  }
};
```

Fig. 10. UVM-SC component instantiating a SystemVerilog UVM component.

The *sc_harness* class in Figure 10 has few additional member fields (compared to those shown on Figure 9). The class member *vip_env* points to *sc_harness*'s hierarchical child component. In the previous section we explained that UVM-ML OA supports a unified logical hierarchy. In the given use case, *env* is a proxy of the SystemVerilog child component of the UVM-SC parent component.

The hierarchical construction is performed in the build phase of *sc_harness*, as prescribed by the UVM methodology. *Vip_env* is assigned with return value of the SystemC ML adapter's method *uvm_ml_create_component*. This method accepts 4 arguments: the string identifier of the target framework (in our case, it is "SV"), the type name of the SystemVerilog component ("comm_vip_env"), instance name ("env") and a pointer to the parent component (*this*).

There is also a TLM analysis port *aport* that is used for communication between the parent and child components. It passes polymorphic transactions of the base class *uvm_seq_control_base*. This and the derived class definitions are not shown here due to the paper's constraints. The binding between the port and its counterpart implementation (export) is done by the ML adapter's method *uvm_ml_connect* that receives two string arguments. One of the benefits of the unified ML hierarchy is that it allows the integrator to use relative names for connection between the parent and child (see *aexport_name* on Figure 10). In our use case, the integrator does not need to keep track of a shadow SystemVerilog hierarchy in SystemC, but only needs to use the child component's export field name "*control_imp*" (as used also in the single-language environment). By using the hierarchical port names, the integrator eliminates any potential scalability issues because those names are guaranteed to be unique always.

The ML adapter's function *uvm_ml_connect()* is invoked in the *connect* phase, synchronized to its corresponding phase in the other connected frameworks. Synchronized phasing ensures that all the ports in all the frameworks are already instantiated prior to being connected. This also allows the backplane to issue a non-delayed error message if the integrator specifies incorrect or invalid port names.

UVM recommends using SystemVerilog virtual interfaces for signal-level access between a verification component (e.g. a driver or a monitor) and the DUT. In our use case, the DUT has two interfaces: *host_if* and *comm_if*. In a single-language environment, the integrator can pass the virtual interface from the DUT to the verification component, using the configuration mechanism. In a multi-language environment, passing the interfaces by pointer is not possible but also not required. Access from a hardware verification language (HVL) to a hardware description language (HDL) is usually implemented via a standard API, for example SystemVerilog VPI or DPI. Those API's operate with the hierarchical names (of signals, functions etc.) in the string format. Consequently, in the ML environment, the UVM-ML integrator needs to pass a hierarchical name of the DUT interface as a string.

Figures 11 and 12 illustrate how the UVM ML configuration mechanism can be used for passing the interface names between frameworks. Figure 11 shows the DUT testbench (the same module *comm_subsystem_tb* as in Figure 7), writing the interface names in the configuration database.

```
// SystemVerilog testbench and DUT

import uvm_pkg::*;

interface comm_if(input logic clk);
  …; string this_name = $sformatf("%m");
endinterface

interface host_if();
  …; string this_name = $sformatf("%m");
endinterface

module comm_subsystem_tb;
  logic clk = 0;
  comm_if cif (clk);
  host_if hif();
  comm_subsystem_top comm(.cif (cif), .hif(hif));
  initial begin
    uvm_config_db#(string)::set(null,
                   "uvm_test_top.*",
                   "comm_intf_name",
                   cif.this_name);
    uvm_config_db#(string)::set(null,
                   "uvm_test_top.*",
                   "host_intf_name",
                   hif.this_name);

  end
endmodule
```

Fig. 11. Passing SystemVerilog DUT interface names in the configuration database.

As previously mentioned in Section III, the SystemVerilog ML adapter broadcasts the UVM configuration settings to the rest of the participating frameworks. Figure 12 shows how the *e* UVC should be extended to retrieve and use the communication interface name.

```
// e env retrieving DUT interface name
unit comm_dut_intf {
  clk_p: in event_port is instance;
    keep clk_p.hdl_path() == "clk";
  data_p: inout simple_port of int is instance;
    keep data_p.hdl_path() == "data"; …
};
unit comm_uvc_env_t {
  dut_intf: comm_dut_intf is instance;

  comm_intf_name: string;
    keep uvm_config_get(comm_intf_name);
  keep dut_intf.hdl_path() == comm_intf_name;
  keep agent.driver.signal_map == dut_intf;
};

// e sequence using ports of comm_dut_intf
extend tx_sequence {
  num_t: int;
  body () @driver.signal_map.clk_p$ is only {
    for j from 1 to num_t {
      wait @driver.signal_map.clk_p$;
      driver.signal_map.data_p$ = j; … };
    };
};
```

Fig. 12. ***e*** UVC retrieving SystemVerilog interface name.

In Figure 12, the UVM *e* unit *comm_uvc_env_t* retrieves the hierarchical interface name using the constraint *keep uvm_config_get*(). This name is propagated to the *e* interface unit *comm_dut_intf* (using an *hdl_path* attribute). In this way, all the *e* ports are automatically associated with the corresponding SystemVerilog interface signal names (because the *hdl_path* attributes of units and ports are implicitly concatenated).

In UVM, settings from hierarchically higher levels made at build time have higher precedence. This feature allows a component on a higher level to override a default configuration specified in a lower-level component. Figure 13 illustrates how the SystemVerilog *comm._vip_env* sets a default configuration for the *e* communication UVC agent (*UVM_PASSIVE*) and how the SystemC testbench harness component provides the actual configuration (*UVM_ACTIVE*) for the agents.

```
// e agent configuration:
<'
unit comm_uvc_agent like uvm_agent {
  keep soft uvm_config_get(active_passive); …
};
'>
…
// SystemVerilog VIP env instantiating the e UVC
// and setting the default configuration for the
// e agent to be passive:
import uvm_pkg::*;
import uvm_ml::*;
class comm_vip_env extends uvm_env;
  …
  `uvm_component_utils(comm_vip_env)
  uvm_component comm_uvc_env;
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(int)::set(this,
      "comm_uvc_env.agent", "active_passive",
      uvm_active_passive_enum'(UVM_PASSIVE));
    comm_uvc_env = uvm_ml_create_component("e",
          "comm_uvc_env_t","comm_uvc_env",this);
  endfunction
  …
endclass
…
// SystemC harness env (see Fig. 10) configures
// the comm UVC agent to be active and thus
// overrides the default
#include "uvm_ml.h"
using namespace uvm_ml;
class sc_harness : public uvm_component {
public:
  void build_phase(uvm_phase *phase) {
    set_config_int("vip_env.*.agent",
      "active_passive",
      uvm_active_passive_enum(UVM_ACTIVE));
    vip_env = uvm_ml_create_component(
              env_config->frmw_name,
              env_config->type_name, "env",this);
  }
  …
};
```

Fig. 13. Hierarchical configuration during the build phase.

V. Future Directions

UVM-ML OA, as developed by AMD and Cadence, was intentionally developed to serve the verification community as a basis for standardization. It is provided as open source under the Apache 2.0 license, and is currently posted on Accellera website [12].

The work presented in this paper was done with awareness and attention to the establishment of the Accellera Multi-Language Working Group (MLWG). The working group provided a broad set of user community requirements for the emerging ML standardized solution, and we strive to keep UVM-ML OA aligned with these requirements.

There are various additional frameworks or enhancements to frameworks that companies or organizations have expressed interest in. We are open to explore collaboration with other framework developers to enable broader integration.

As previously mentioned, UVM-ML OA currently includes ML enabling reference patches for the standard frameworks (UVM SystemVerilog and ASI SystemC). If UVM-ML OA is adopted as the basis for standardization by the Accellera ML WG, then we intend to work with the corresponding owning Accellera working groups to incorporate the ML specific requirements and address them in the future releases of their corresponding frameworks, thus eliminating any provided patches.

UVM-ML OA will continue to be developed. Currently it supports a rich set of facilities that enable the UVM methodology for ML, however there are still some important features missing and they should be added in the near future. This includes the following ML capabilities:

- Coordinated test framework completion and shutdown
- ML messaging service
- Error reporting and handling
- Time quantum service
- Basic UVM synchronization facilities (events, objections, barriers) working for ML
- Sequence coordination between frameworks
- Expanded set of supported transaction types in the TLM communication
- Enhanced debugging capabilities and tracing
- Expanded methodology guidelines and examples addressing the high-level verification tasks

Summary

This paper describes the challenge of integration and reuse of verification components based on different methodologies and languages when combined together into a single environment. Some key relevant use cases described in this paper include VIP reuse, multi-abstraction substitution, hardware assisted verification, and software driven verification. These use cases were presented to illustrate the broad spectrum of applications that face this common challenge.

The paper presents an overview of a novel solution called UVM-ML OA. UVM-ML OA represents a step forward in technology and a generic approach to enabling multiple frameworks to interoperate within the same environment. The benefits of this architecture were described as well. UVM-ML OA is capable of addressing all of the use-cases presented and beyond, without any underlying assumptions about the number or types of frameworks that are integrated. An in-depth illustrative use-case example was then provided to demonstrate the capability and flexibility of the UVM-ML OA solution.

The UVM-ML OA solution will continue to evolve and align with emerging industry needs. It contains a rich set of facilities and will be further enhanced to broaden its capabilities.

We believe that UVM-ML OA will be extremely beneficial for users facing practical multi-framework interoperability challenges and will provide them with an efficient and well architected solution. We also hope that it can become a basis for the emerging standardization in this area.

References

[1] B. Bailey, F. Balarin, M. McNamara, G. Mosenson, M. Stellfox, Y. Watanabe, “TLM-Driven design and verification methodology”, Cadence Design Systems, Inc., 6-12, June 2010, http://www.amazon.com/TLM-Driven-Design-Verification-Methodology-Bailey-ebook/dp/B003XVZBE8.

[2] A. Jain, G. Bonanno, Dr. H. Gupta, A. Goyal, A. Mangla. "System Verilog Universal Verification Methodology based verification environment for imaging IPs/SoCs", CDNLive Users Conference, Bangalore, 2012, http://www.cadence.com/cdnlive.

[3] S. Swan, Qiang Zhu, Xingri Li, “Moving to SystemC TLM for design and verification of digital hardware” EE Times, May 13, 2013, http://www.eetimes.com/document.asp?doc_id=1280845.

[4] K. Herterich, " Verifying Multiple DUV Representations with a Single UVM-e Testbench", CDNLive Users Conference, Boston MA, August 2013, http://www.cadence.com/cdnlive.

[5] Srivatsan Raghavan, “Can hardware-assisted verification save SoC realization time?", EE Times, November 8, 2013 http://www.eetimes.com/author.asp?section_id=36&doc_id=1320016.

[6] Sumeet Aggarwal, “Rapid Adoption Kit (RAK) -- creating UVM verification environments with hardware-assisted verification,” System Design and Verification Blog, Cadence Design Systems, June 28, 2013 http://www.cadence.com/Community/themes/blogs.

[7] D. Black, J. Aynsley, B. Bunton, V. Essen, T. Wieman, Technical Tutorial: "Software-driven verification using TLM-2.0 virtual platforms". Design and Verification Conference (DVCon) February 28, 2011, http://videos.accellera.org/tlm20sdvvirtual/ts82dpj34d/index.html.

[8] Sandeep Jana, Sonik Sanchdeva, Krishna Kumar, Swami Venkatesan, Debajyoti Mukherjee, "TLM based software control of UVCs for Vertical Verification Reuse", DVClub, Apr 18, 2013, http://www.slideshare.net/directory/slideshows/tlm-based-software-control-of-uvcs-for-vertical-verification-reuse.

[9] J. Aynsley, "VMM-to-SystemC Communication Using the TLI", VMM Central, http://www.vmmcentral.org/vmartialarts/2011/03/vmm-to-systemc-communication-using-the-tli.

[10] A. Erickson, "Transaction-Level Friending: An Open-Source, Standards-Based Library for Connecting TLM Models in SystemC and SystemVerilog", DVCon 2013, San Jose, CA http://events.dvcon.org/2013/proceedings/papers/07_1.pdf.

[11] Multi-Language (ML) Working Group http://www.accellera.org/activities/committees/multi_language.

[12] UVM-ML Open Architecture version 1.3 (Overview and download) http://forums.accellera.org/files/file/65-uvm-ml-open-architecture.

[13] B. Sniderman, V. Yankelevich, “Techtorial: UVM Multi-Language: Technology and Reference Application”, CDNLive Users Conference, Santa Clara CA, March 2013, http://www.cadence.com/cdnlive/na/2013/Pages/default.aspx.

[14] Shankar Myilswamy, Kai Chirca, Matthew Pierson, Kedar Basavaraj, Izuchukwu Nwachukwu, " Molding multi-methodology (VMM, UVM, SystemC) VIPs for reuse", CDNLive Users Conference, Boston MA, August 2013, http://www.cadence.com/cdnlive, pp.11,17.

[15] G. Leshem, V. Yankelevich, B. Sniderman. "UVM-ML Whitepaper. A Modular Approach for Integrating Verification Frameworks" (available for download at http://forums.accellera.org/files/file/65-uvm-ml-open-architecture.

[16] UVM-ML Integrator User Guide (available for download at http://forums.accellera.org/files/file/65-uvm-ml-open-architecture.