Multi-Language Verification: Solutions for Real World Problems

By
Agenda

• Multi Language (ML) Verification Challenges
• Use Cases Requiring an ML Solution
• UVM-ML Open Architecture (OA) – An Overview
• Unified Hierarchy and Phase Alignment
• An Illustrative Use Case Detailed
• Summary
ML Verification Challenges

• The need to deal with several implementation languages and diverse verification methodologies is only one of the subsystem - and system – level verification challenges
• SystemVerilog, e, SystemC and C++ are commonly used for verification purposes
• UVM, OVM and VMM libraries are used for development of reusable testbenches
• Adoption of external verification IP’s consistently increases
• Difficulty of integration may impact cost and time to market
• The need to reuse one useful VIP, implemented in a different language, is a sufficient reason to necessitate a multi-language environment
• Multiple Design abstractions in different languages can be employed
• Start with high-level SystemC (SC) model
• Progress to refined RTL and re-use SC model abstraction concurrently
Hardware-Assisted Design Acceleration Use Case

- Accelerated DUT abstraction for verification acceleration
- Specialized case of multi-abstraction use case
- Re-use same testbench with minimal changes
Software Driven Functional Verification Use Case

- Leverage software to drive ‘real world’ verification scenarios
- Software environment may be deployed in one language while simulated frameworks are in another languages
- Necessitates a multi-language solution
UVM Multi-Language Open Architecture

• UVM-ML OA was developed jointly by Cadence and AMD and is available as open source under the Apache 2.0 license

• It is posted on the Accellera website at http://forums.accellera.org/files/file/65-uvm-ml-open-architecture

• It was intentionally developed to serve the verification community as a basis for standardization

• The presented work was done with attention to the establishment and requirements of the Accellera Multi-Language Working Group
UVM-ML OA Key Features

- Framework- and simulator– independent API
- Coordinated initialization
- Delegation of system services
- Pre-/Post-/Runtime phase synchronization
- TLM communication (TLM1 and TLM2)
- Unified hierarchy solution
- ML configuration
- Broader synchronization support
UVM-ML OA Overview

Backplane serves as a central hub connecting the frameworks in a star topology (not peer-to-peer).

Each framework is connected via an adapter to the backplane using the same API.

System services may be provided by frameworks or as dedicated services.

UVM-ML BACKPLANE

Registration API
Service API
Provided API
Required API

Framework / Routing / Proxy Info
Unified Hierarchy and Phases

ML Unified Hierarchy

Phase Sequence Representation

- Depth-First Build by Hierarchy
- Top-down ‘build’ pre-run phase example
An Illustrative Use Case

• The use case represents an ML verification environment for a hypothetical communication subsystem

• A SystemVerilog DUT testbench instantiates a module and its two interfaces

• The verification environment for this testbench is composed of three layers:
  – SystemC layer that can be controlled by an application SW
  – SystemVerilog subsystem-level VIP
  – Interface UVC’s implemented in UVM-SV and UVM-e
An Illustrative Use Case: Hierarchical Diagram

Test Harness Layer

- SystemC
  - Ref model

- SystemVerilog
  - Comm UVC env
  - Scoreboard
  - Host UVC env
  - Virtual sequencer
  - Command

Subsytem-level VIP

- Comm UVC
  - vip_env_t
  - config_db
  - Test
  - Command

DUT TB

- COMM_SUBSYSTEM_TB
  - 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", cif.this_name)

DUT

- COMM_SUBSYSTEM
  - HOST_IF
  - COMM_IF
```cpp
#include "uvm.h"
#include "uvm_ml.h"

class sc_harness : public uvm_component {
    public:
        command_api ca; // command API
        void build_phase (uvm_phase *phase) { ... }  
        void run_phase (uvm_phase *phase) { ... }  
        UVM_COMPONENT_UTILS(sc_tb) ...
    
    class test1 : public uvm_component  
    { // Mimicking SW-driven test
        sc_harness * sc_h;
        void build_phase (uvm_phase * phase) { ... }  
        void run_phase (uvm_phase *phase) {  ...  
            VIP_ENV_header
            Connecting ML TLM using relative names
        
        // Instantiating foreign component
        uvm_component * vip_env;
        tlm_analysis_port<uvm_seq_control_base> aport;
        sc_harness(sc_module_name nm):
            uvm_component(nm)
        { ...; uvm_ml_register(&aport); }  
        void build_phase(uvm_phase *phase) {  ...  
            vip_env = uvm_ml_create_component (  
                env_config->frmw_name,  // “SV”  
                env_config->type_name,  // “vip_env_t”  
                “vip_env”, this);  
            ...  
            void connect_phase (uvm_phase *phase) {  
                string aexport_name =  
                    vip_env->name()+string(".")+"control_imp";  
                uvm_ml_connect(aport.name(), aexport_name);  
            }
    }
}
```
Instantiating and configuring interface UVC's

```
import uvm_pkg::*;
import uvm_ml::*;
class comm_vip_env extends uvm_env;
...
uvm_component comm_uvc_env;
uvm_component host_uvc_env;
function void build_phase(uvm_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);
    host_uvc_env = host_uvc_env_t::type_id::create ("host_uvc_env", this);
endfunction
endclass
```

```
// 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;
    agent: uvm_agent is instance;
    comm_intf_name: string;
    keep uvm_config_get(comm_intf_name);
    keep dut_intf.hdl_path() == comm_intf_name;
 ...
};
>'
```
Summary

• UVM-ML OA represents a generic approach to enabling multiple frameworks to interoperate within the same environment.

• UVM-ML OA is capable of addressing the use-cases presented and beyond, without any underlying assumptions about the number or types of language and methodology frameworks that are integrated.

• It will continue to evolve and align with emerging industry needs.