Using UVM-ML library to enable reuse of TLM2.0 models in UVM test benches

Sarmad Dahir (sarmad@cadence.com)
Cadence Design Systems

DVCON Europe 2018

cadence®

Hardware Design and Verification Challenges

Growing Verification Challenges

- As HW design complexity is increasing, the verification part of the ASIC development flow is also becoming ever more challenging and important.

- Advanced verification methodologies like metric-driven verification are used to address these challenges
  - ASIC verifiers rely on using UVCs and reference models, from different sources, to build their verification environments.
  - Both Loosely-Timed TLM 2.0 models as well as UVM make use of sockets/ports, but they cannot be directly connected to each other, which adds a limitation on the reusability of the reference models.
  - To overcome the above limitation, verifiers need to manually implement an additional ad-hoc wrapper layer that enables the communication between TLM and UVM, e.g. by using DPI-C.
  - A more generic solution is to use a library, like UVM-ML, that enables direct communication between different frameworks.

TLM 2.0 Models and UVM Multi-Language Library

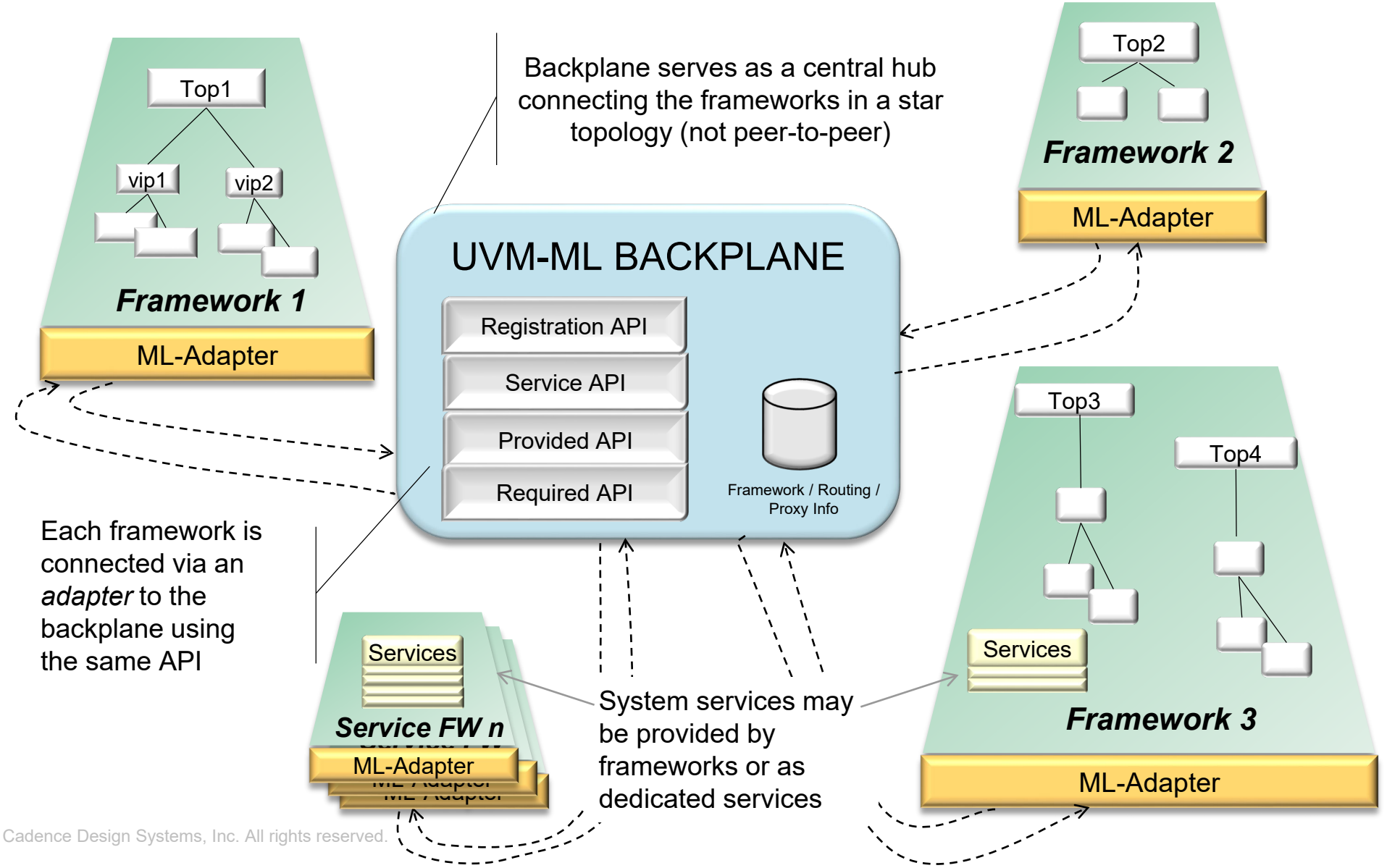
- The hardware design flow usually starts with an early virtual prototypes development phase where reference models are developed to explore different aspects of the target algorithms, e.g. performance, accuracy and architecture complexity.
  - These virtual prototypes are in some cases made available before the hardware design phase starts and are used as early software development platforms.
- TLM 2.0 models are a natural fit to use with UVM testbenches because both UVM, as well as TLM and SystemC®, make use of sockets, ports, etc. and abstract data representation.
- To leverage all available types of UVCs and HW reference models in verification testbenches users need to have a flexible and easy to use environment where these different components can co-exist and work seamlessly together.
- By using UVM ML, verifiers can reuse the TLM 2.0 models in their verification environments by directly connecting the ports of the testbench to the ports of the TLM model.

Introduction to UVM Multi-Language

UVM Multi-Language Development

- The UVM Multi-Language library was developed by a collaboration between Cadence and AMD
- The people and the library were the base for the Accellera UVM-ML Open Architecture (OA) working group
- UVM-ML library is available as open source (since July 2012)
  - http://forums.accellera.org/files/file/65-uvm-ml-open-architecture
  - And it's also provided within Cadence® Xcelium™ installations

UVM-ML OA Architecture

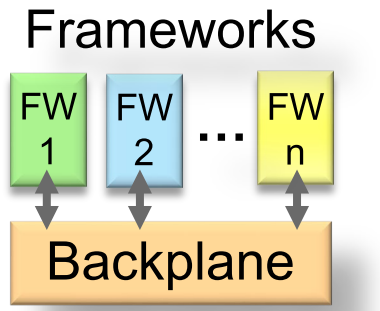


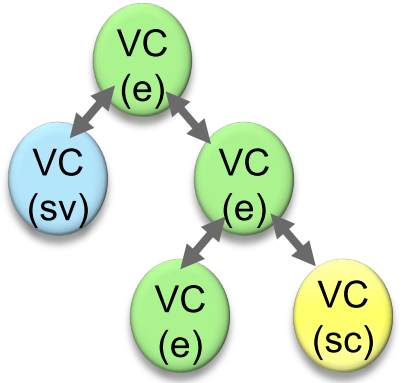
UVM-ML OA Key Features

- Modular and extensible architecture
- Framework and simulator independent API
- TLM communication (TLM1 and TLM2)

- Coordinated initialization/bring-up of all participating libraries
- Phase synchronization

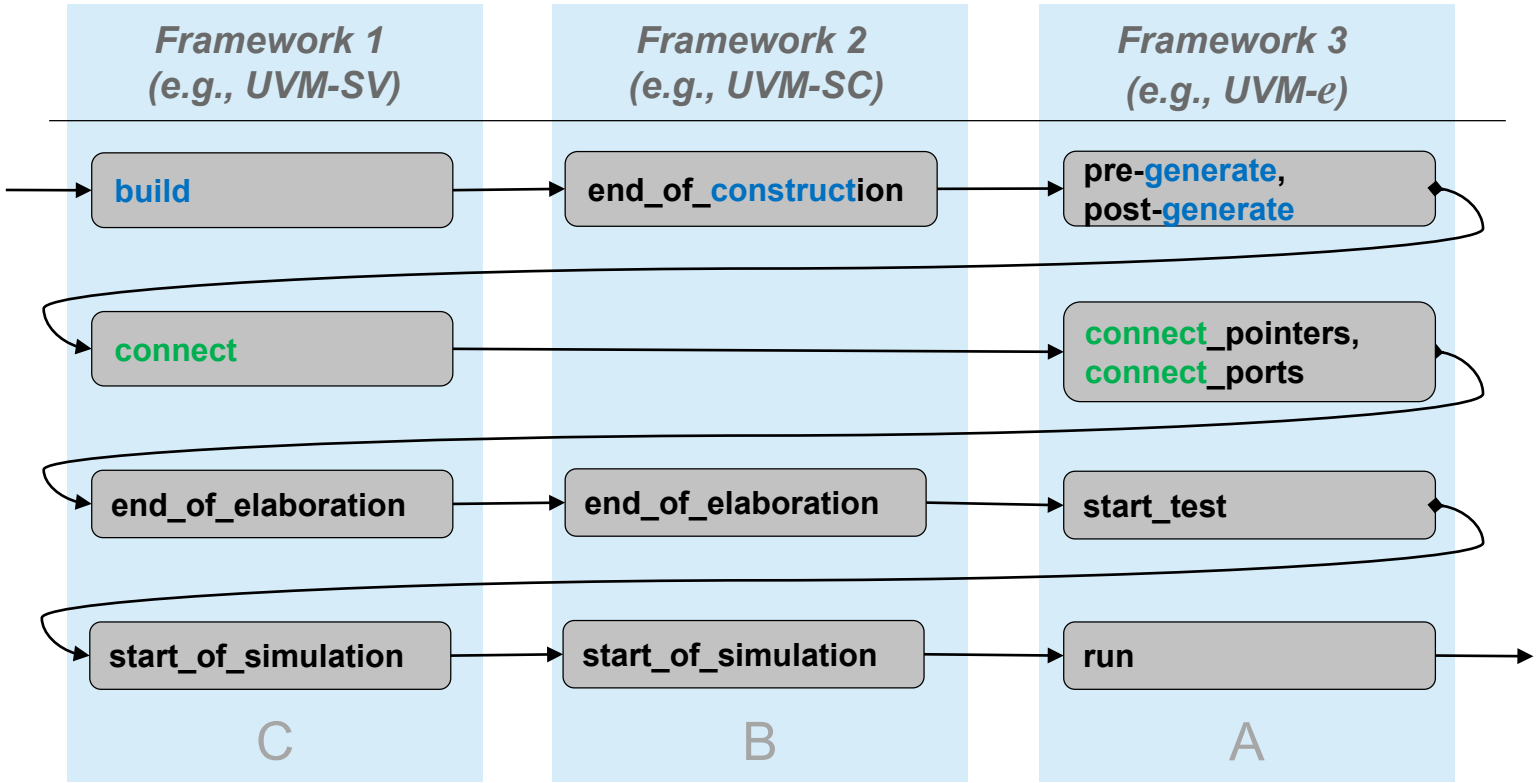
- Unified hierarchy solution:
  Hierarchical construction of a multi-language verification environment
- Multi-language configuration
- Multi-language sequence layering
- Multi-language debugging/synchronization





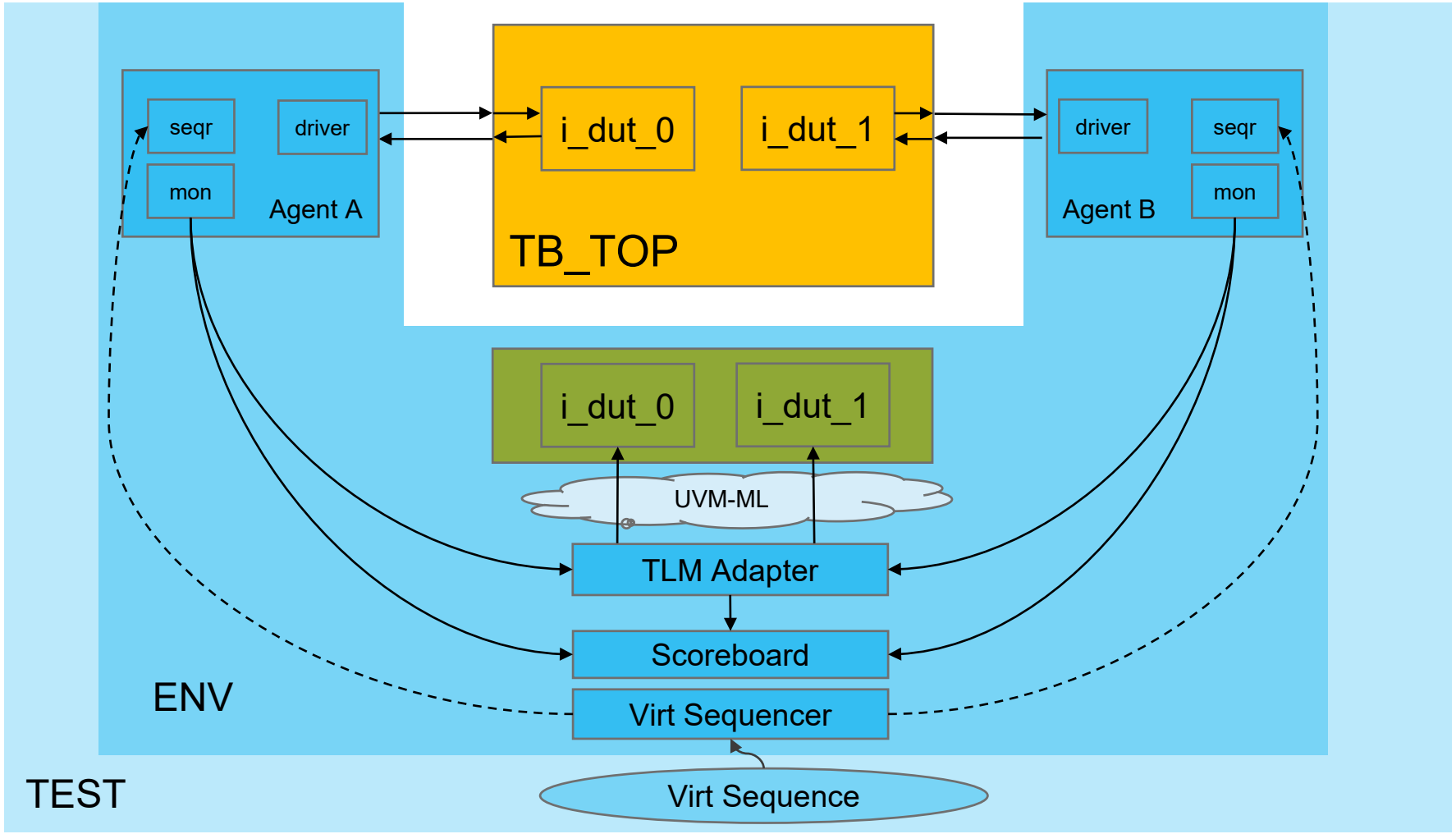
Synchronized Phases

- Coordinate phases between the various frameworks
- Ensures that components are ***configured*** then ***created***, ports ***created*** then ***connected***, etc.



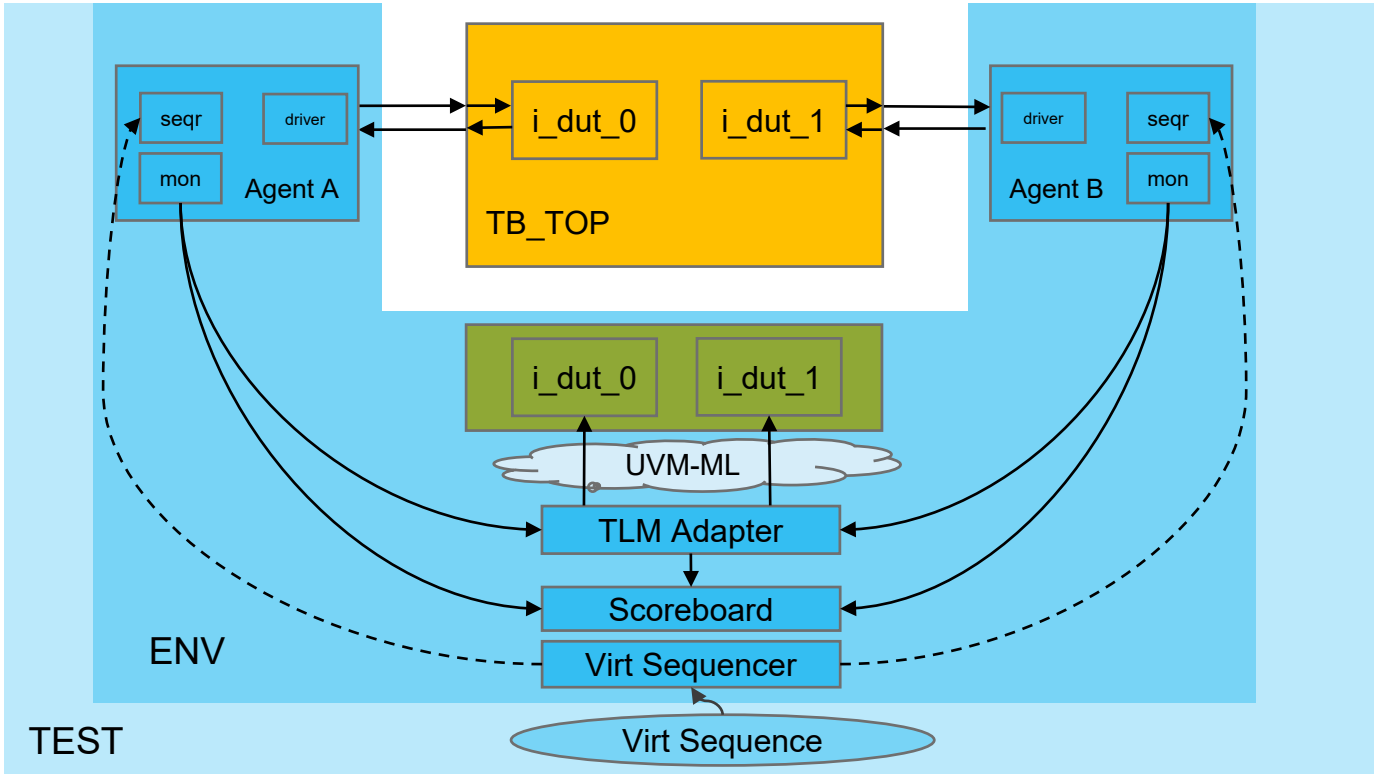
Example of Using UVM-ML in a UVM+TLM Testbench

Topology of Verification TB when reusing the TLM Model



Comparing HDL vs TLM in the Verification Testbench

- The UVC monitor passes the observed transactions to the TLM adapter.
- The TLM adapter maps UVM sequence items to TLM GP and sends them, over UVM-ML, to the TLM model.
- The response from TLM is passed to the scoreboard through the TLM adapter.
- This way the testbench scoreboard can compare the functionality of RTL to the TLM reference model.



Unified Hierarchy – Instantiating a SystemC Component Under UVM-SV Component

- Each type must be defined in the respective language
- In the SV component:
  - **Define** a member of type **child_component_proxy** (or **uvm_component**, which it inherits)
  - **Create** it using **child_component_proxy uvm_ml::uvm_ml_create_component()**

SC

```
class top_wrapper : public uvm_component {}
```

SV

```
class myenv extends uvm_env;
  uvm_component tlm_wrapper_inst;

 function void build_phase(uvm_phase phase);
   …
   tlm_wrapper_inst = uvm_ml_create_component("SC",
                                              "top_wrapper",
                                              "tlm_wrapper_inst",
                                              this);

  endfunction
  …
```

Foreign Type

Instance Name

Parent

UVM-ML Integration Step by Step

Importing the UVM-ML Library

In addition to the RTL and testbench, users need to load/compile the backplane + adapters in order to use UVM-ML features

In SystemVerilog:

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import uvm_ml::*;
```

In SystemC®:

```
#include "uvm_ml.h"
#include "ml_tlm2.h"
```

Running the Test Under UVM-ML

When incorporating UVM-ML in your environment, you need to let UVM-ML take control over building your UVM environment and running its phases

Instead of using the regular UVM “run_test()” method, use the UVM-ML variant “uvm_ml_run_test(string)”:

```
string tops[1];
initial begin
  …
  tops[0] = "";
  uvm_ml_run_test(tops);
  …
```

The test name can still be passed using the plus argument +UVM_TESTNAME to the xrun command

Socket Registration on SV Side

Using UVM-ML “register” API, TLM sockets are registered for connection to SC domain.

UVM-ML Registration:

```
function void phase_ended(uvm_phase phase);
    if (phase.get_name() == "build") begin
        uvm_ml::ml_tlm2#()::register(initiator_socket_0);
        uvm_ml::ml_tlm2#()::register(initiator_socket_1);
    end
  endfunction
```

The registration should be done at phases_ended() of ***build*** phase as shown above

Socket Registration on SystemC Side

The sockets registration needs to be done in SystemC® side also, which can be done by using the socket registration macros:

```
void build() {
  full_target_socket_name_0 = ML_TLM2_REGISTER_TARGET(dut_inst, tlm_generic_payload, reg_in_0, 32);
  full_target_socket_name_1 = ML_TLM2_REGISTER_TARGET(dut_inst, tlm_generic_payload, reg_in_1, 32);
}
```

Notes:

1. The macros above return a string value, which actually is the hierarchical path to the socket after elaboration
   - this can be printed out for debugging
2. Instead of using the macros, you can also call the socket registration functions directly in order to take advantage of all additional arguments provided:
   ml_tlm2_register_initiator <tlm_generic_payload, 32>(top_inst.dut_i, top_inst.dut_i.o_port[0], "o_port_0");

Socket Connection

The connection from SV to SystemC ® domain can be done in either the SV or SystemC ® side; here it is done in SV side in the connect_phase() function:

```
function void connect_phase(uvm_phase phase);
   void'(uvm_ml::connect(initiator_socket_0.get_full_name(), "uvm_test_top.tb_env.tlm_wrapper_inst.dut_inst.reg_in_0"));
   void'(uvm_ml::connect(initiator_socket_1.get_full_name(), "uvm_test_top.tb_env.tlm_wrapper_inst.dut_inst.reg_in_1"));
endfunction
```

The SystemC target sockets names could be extracted either by looking into the topology of the testbench or using the return value of SystemC socket registration as was discussed in previous slide

Using TLM Generic Payload Extensions with UVM-ML

Generic Payload Extension Updates on SystemVerilog Side

The class that defines the GP extension fields must use the UVM filed registration macros which define the packer and unpacker functions under-the-hood; these functions are used by UVM-ML library for serialization and de-serialization of GP for transporting and collecting data:

```
`uvm_object_utils_begin(vip_transfer_gp_ext)
   `uvm_field_int(m_id, UVM_ALL_ON)
 `uvm_object_utils_end
```

Also, the names of the GP extension classes in SV and SystemC® side should match; if they don't, then specify the matches using the set_type_match() function:

```
if(!uvm_ml::set_type_match("sv:vip_transfer_gp_ext","sc:transfer_gp_ext"))
   `uvm_fatal(get_name(), "Cannot match the GP extension types.")
```

The above function call tells UVM-ML that the SV class "vip_transfer_gp_ext", which is the class that specifies the GP extension, should be considered the corresponding match for the SC class "transfer_gp_ext"

Generic Payload Extension Updates on SystemC Side

The field registration macros are needed in SystemC® side too; on the SystemC side, they are out-of-class macros. This enables code reuse as these macros can be added in a top TB file only when used in a TB that needs to send this class type across different frameworks:

```
ML_TLM2_GP_EXT_BEGIN(transfer_gp_ext)
  ML_TLM2_FIELD_ACCESSORS(uint32_t, get_id, set_id)
ML_TLM2_GP_EXT_END(transfer_gp_ext)
```

Notes:

1. Here we used the “ACCESSORS” macro that uses the get_id()/set_id() member functions of the GP extension class to access the data field, which is because the data members are declared private; if they were public then we could use the macro “ML_TLM2_FIELD” that takes the data member as argument
2. The order of field registration macros should match in both SV and SystemC sides for the data to be unpacked properly

Using the Generic Payload Extension on SystemVerilog & SystemC

Using the GP extensions is done as per regular TLM flow; no UVM-ML specific APIs are required

In both UVM-SV and SystemC®, use the ***set_extension()*** and ***get_extension()*** functions to add/get the GP-extension object to/from the GP

UVM-ML Integration Debugging Helpers

1. You can add the following function to call in your SC code before any socket registration, to get the list of registered sockets in both SystemC and SV:
   void(uvm_ml_execute_command("uvm_ml_trace_register -on"));

2. You can also get the list of registered sockets by using this Xcelium™ TCL command:
   uvm_ml trace_register_tlm

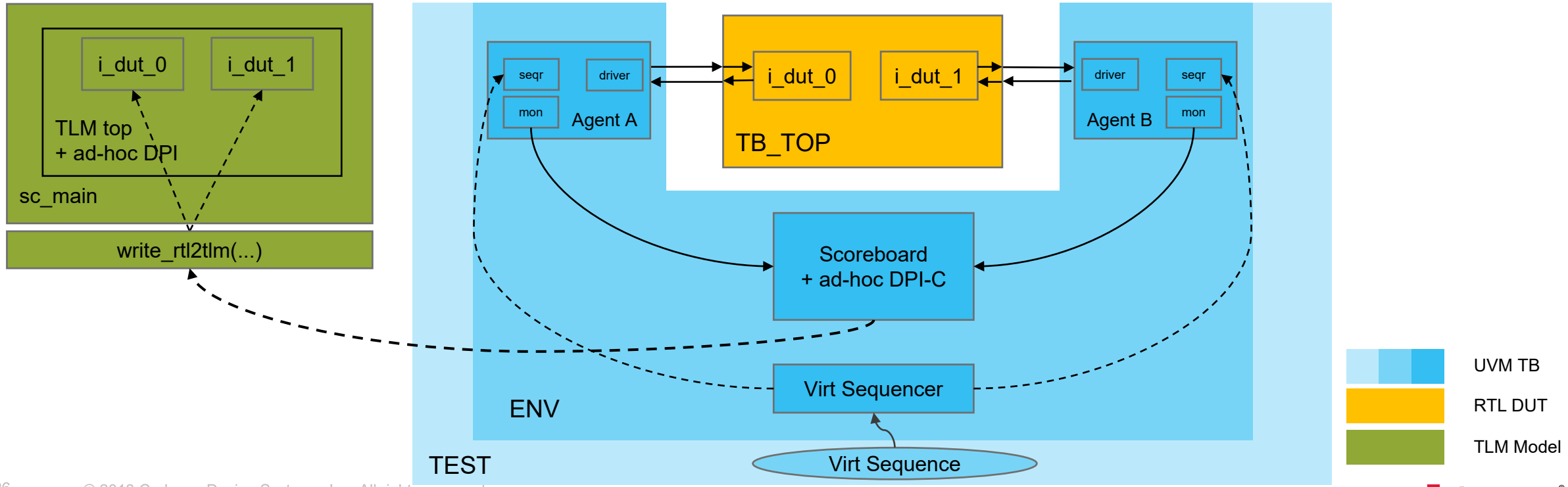
3. To stop the simulation at the end of the connect phase and print the connections to make sure they've been done properly, use the following Xcelium™ TCL commands:
   uvm_ml_phase -stop_at -end connect
   run
   uvm_ml_print_connections

4. The following TCL command shows the types being matched between the different frameworks:
   uvm_ml print_type_match

Discussion and summary

Exploring alternative solutions to UVM-ML

- We also implemented a version of the TB that uses DPI-C calls, instead of using UVM-ML library, to communicate between UVM-SV and SystemC domains.
- Simulation time and memory usage were slightly lower in this version but the code complexity was significantly higher.
  - The code had to include ad-hoc pieces that limit code re-usability.
  - Missing UVM-ML features such as auto conversion of class types, list registered/connected sockets, phase synchronization, etc.



Summary

- UVM-ML is an easy and straight forward approach to integrate and connect TLM models into a MDV environment.
  - The complexity of the UVM scoreboard component was significantly lower as we didn't need to re-model the intended DUT functionality to support the self-checking test bench approach.
  - Reduced TB development time by leveraging on code reuse.
  - The reduction of the test bench complexity did not cause any decrease in verification quality .
  - It offers advanced and convenient features like unified hierarchy and phase synchronization.
- Allows direct connection of TLM models and UVM test benches by reusing already existing sockets, ports.
- Open source library which is simulator independent.
  - Can be downloaded from Accellera and is also delivered with Xcelium™ simulator.
- Cadence UVM-ML support team is ready to help if needed:
  - support_uvm_ml@cadence.com