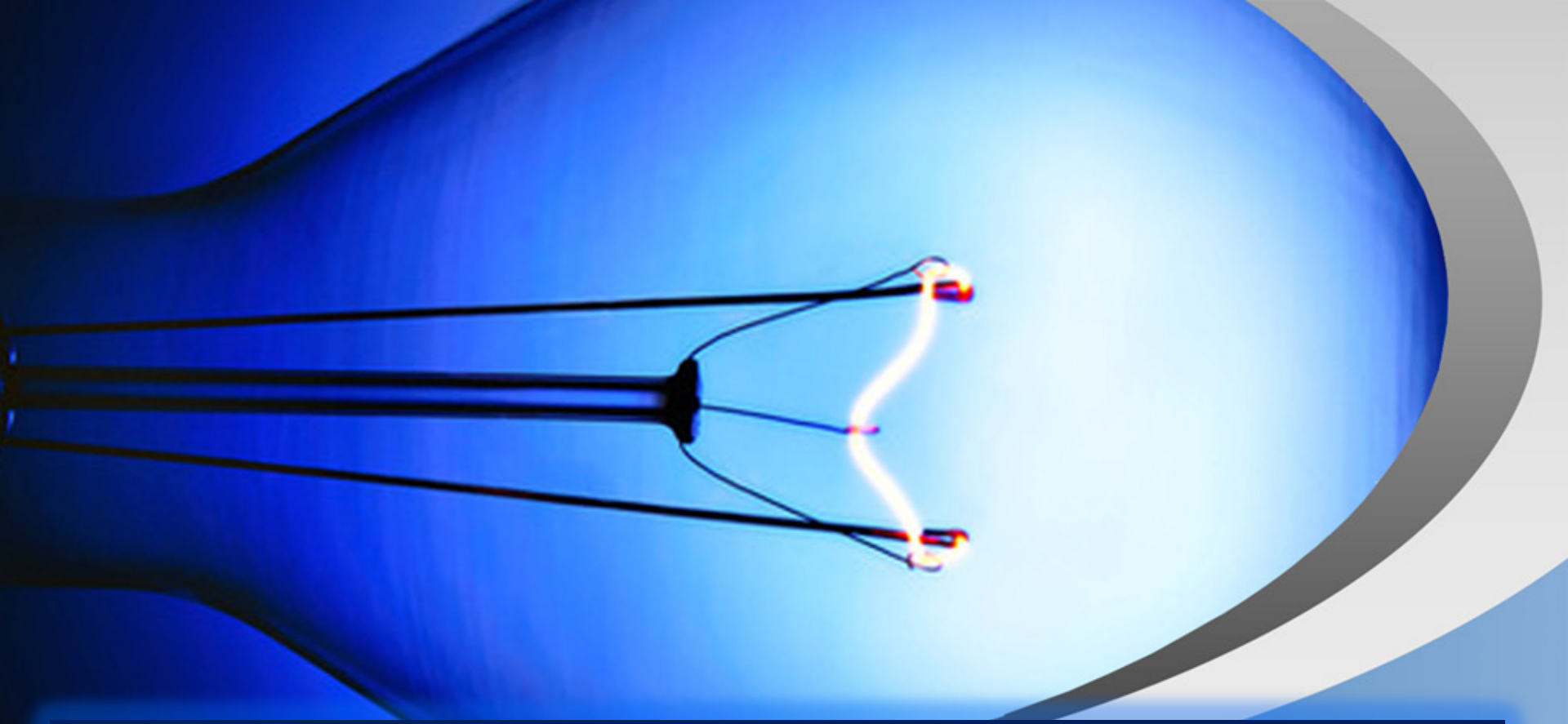


Lessons from the Trenches: Migrating Legacy Verification Environments to UVM™

Tutorial presented by members of the VIP TSC

DVCon 2013
Design & Verification Conference & Exhibition

UVM

accellera
SYSTEMS INITIATIVE

Agenda

- **Anecdotes From Hundreds of UVM Adopters**
  - John Aynsley, Doulos
- **Migrating from OVM to UVM A Case Study**
  - Hassan Shehab, Intel
- **A Reusable Verification Testbench Architecture Supporting C/UVM Mixed Tests**
  - Richard Tseng, Qualcomm
- **UVM to Rescue – Path to Robust Verification**
  - Asad Khan, Texas Instruments
- **OVM to UVM Migration**
  There and Back Again, a Consultant's Tale
  - Mark Litterick, Verilab
- **IBM Recommendations for OVM → UVM Migration**
  - Wes Queen, IBM
- **FPGA chip verification using UVM**
  - Charles Zhang & Ravi Ram, Paradigm
- **Please ask questions during/end of each talk**
- **Fill out survey questionnaire**

Anecdotes From Hundreds of UVM Adopters

John Aynsley

General Comments

Especially managers and self-teachers

- **Underestimating the learning curve**
- **Directed tests versus constrained random verification**
- **Reuse and OOP expertise**
- **UVM only gets you so far**

OVM to UVM

- There exists plenty of guidance on migrating from OVM to UVM
- www.uvmworld.org

  ovm2uvm_migration.pdf
- verificationacademy.com/verification-methodology
- http://www.doulos.com/knowhow/sysverilog/uvm/ovm-to-uvm

General Issues with UVM

Studying the documentation is not enough!

- Certain UVM concepts are not straightforward
- SV/UVM terminology can be a barrier
- The sheer size of the UVM BCL
- Too much choice

Backward compatibility with legacy

- Lack of recommended practice and naming conventions

Top UVM Time-Wasters

- Field macros
- `uvm_do macros
- Deprecated OVM sequence mechanisms
- Confusion over the config db (and the OVM legacy)

Those Evil Field Macros?

```
class basic_transaction extends uvm_sequence_item;

  rand bit[7:0] addr, data;

  function new (string name = "");
    super.new(name);
  endfunction: new

  `uvm_object_utils_begin(basic_transaction)
    `uvm_field_int(addr, UVM_DEFAULT)
    `uvm_field_int(data, UVM_BIN | UVM_NOCOPY)
  `uvm_object_utils_end

endclass : basic_transaction
```

Field Macro Flags

Inclusion in operations

`UVM_DEFAULT`	all on
`UVM_COPY`	default
`UVM_COMPARE`	
`UVM_PRINT`	
`UVM_RECORD`	
`UVM_PACK`	
`UVM_NOCOPY`	need to set explicitly
`UVM_NOCOMPARE`	
`UVM_NOPRINT`	
`UVM_NORECORD`	
`UVM_NOPACK`	
`UVM_READONLY`	not configured

Overriding do_compare

```
class bus_xact extends uvm_sequence_item;
  ...
  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    bus_xact t;
    bit result = 1;
    $cast(t, rhs);

    result &=   comparer.compare_field("op",   op, t.op, $bits(op));
    if (op != NOP)
      result &= comparer.compare_field("addr", addr, t.addr,
                                       $bits(addr));
    ...
    return result;
  endfunction

  `uvm_object_utils_begin(bus_xact)
    `uvm_field_int(op,   UVM_NOCOMPARE)
    `uvm_field_int(addr, UVM_NOCOMPARE)
    ...
  `uvm_object_utils_end
endclass
```

tx1.compare(tx2)

Collects mismatches

Turn off default comparison

Also uvm_packer, uvm_recorder, ...

Field Macros and Overridden Methods

```
uvm_comparer comparer = new;
comparer.policy = UVM_SHALLOW;
comparer.show_max = 999;
tx1.compare(tx2, comparer);
```

Pseudo-code

```
begin
  bit result = 1;
  result &= tx1.field_automation(tx2);
  result &= tx1.do_compare(tx2);
  output_mismatch_report;
  return result;
end
```

UVM_COMPARE
UVM_NOCOMPARE
UVM_REFERENCE

Stop Faffing Around!

```
class basic_transaction extends uvm_sequence_item;

  `uvm_object_utils(basic_transaction)

  ...

  function bit do_compare(uvm_object   rhs,
                          uvm_comparer comparer);
    bit result = 1;
    basic_transaction tx;
    $cast(tx, rhs);

    result &= (addr == tx.addr);
    result &= (data == tx.data);

    return result;
  endfunction

endclass : basic_transaction
```

The Dreaded super.build_phase

```
uvm_config_db#(int)::set(this, "m_env.m_driv", "count", 999);
```

```
int count;


`uvm_component_utils_begin(my_component)
  `uvm_field_int(count, UVM_DEFAULT)
`uvm_component_utils_end


function void build_phase(uvm_phase phase);
  super.build_phase(phase);
endfunction
```

Sets count = 999

Calls apply_config_settings

The Moderately Evil `uvm_do

```
`uvm_do(req)
```

```
req = tx_type::type_id::create("req");

start_item(req);

if( !req.randomize() ) `uvm_error(...)

finish_item(req);
```

Equivalent?

Expanded Invocation of `uvm_do

`uvm_do(SEQ_OR_ITEM)

```
  begin
  uvm_sequence_base __seq;
  begin
  uvm_object_wrapper w_;
  w_ = SEQ_OR_ITEM.get_type();
  $cast(SEQ_OR_ITEM , create_item(w_, m_sequencer, `"SEQ_OR_ITEM`"));
  end
  if (!$cast(__seq,SEQ_OR_ITEM)) start_item(SEQ_OR_ITEM, -1);
  if ((__seq == null || !__seq.do_not_randomize) && !SEQ_OR_ITEM.randomize()
with {} ) begin
    `uvm_warning("RNDFLD", "Randomization failed in uvm_do_with action")
  end
  if (!$cast(__seq,SEQ_OR_ITEM)) finish_item(SEQ_OR_ITEM, -1);
  else __seq.start(m_sequencer, this, -1, 0);
  end
```

The OVM Sequencer Library

```
class my_sequencer extends ovm_sequencer #(basic_transaction);

  `ovm_sequencer_utils(my_sequencer)

  function new(string name, ovm_component parent);
    super.new(name,parent);
   `ovm_update_sequence_lib_and_item(basic_transaction)
  endfunction : new

endclass: my_sequencer
```

Populates sequence lib with simple, random, & exhaustive sequences

Deprecated in UVM

The OVM Sequence

```
class my_sequence extends ovm_sequence #(instruction);

  ...

  function new(string name = "");
    super.new(name);
  endfunction: new

  task body;
    ...
  endtask

  `ovm_sequence_utils(my_sequence, my_sequencer)

endclass: my_sequence
```

Deprecated in UVM

Selecting a Sequence in OVM

```
set_config_string("*.m_seqr", "default_sequence", "my_sequence2");
```

```
set_config_string("*.m_seqr", "count", 10);
```

All firmly deprecated in UVM

Starting a Sequence in UVM

```
sequence.start(sequencer, parent_sequence, priority);
```

Draft UVM Sequence Library

```
class my_seq_lib extends uvm_sequence_library #(my_tx);
  `uvm_object_utils(my_seq_lib)
  `uvm_sequence_library_utils(my_seq_lib)

  function new(string name = "");
    super.new(name);
    init_sequence_library();
  endfunction
endclass
```

```
my_seq_lib lib = my_seq_lib::type_id::create();
lib.add_sequence( seq1::get_type() );
lib.add_sequence( seq2::get_type() );
lib.selection_mode = UVM_SEQ_LIB_RAND;
if ( !lib.randomize() ) ...
lib.start(m_env.m_seqr);
```

Other Detailed UVM Issues

- Need for run-time phasing
- Confusion over when to raise/drop objections
- Register layer seems hard

- How to handle layered sequencers and agents
- Confusion over the semantics of lock/grab

Session 8: Hardcore UVM - I , Weds 10:30am – 12:00pm

The Finer Points of UVM: Tasting Tips for the Connoisseur (myself)

Beyond UVM: Creating Truly Reusable Protocol Layering (Janick)

Things Missing from UVM

- Mixed language support (RTL)
- Mixed language support (TLM)
- Using UVM with analog/AMS

Migrating from OVM to UVM
A Case Study

Hassan Shehab

Technical Validation Lead

Intel Corporation

Agenda

- We present a OVM compatibility layer on top of UVM that allows the use of OVM based IPs on UVM source code
- We look at the results of using the compatibility layer by migrating a SoC consisting of 25+ OVM VIPs

Introduction

- **We present a case study of migrating a SoC environment fully developed on OVM to UVM**
  - UVM recommends running a converter script on the source code to replace the ovm_* symbols with uvm_* symbols
  - This mandates either abandoning the OVM code base of the VIPs or maintaining two repositories
  - With heavy OVM in use, this is NOT practical as VIPs needs to go into SoCs with OVM base and UVM base running in parallel.

- **Enhanced the OVM compatibility layer developed originally by Mark Glasser part of UVM EA**
  - https://forum.verificationacademy.com/forum/uvmovm-kit-downloads-and-user-contributions-forum/kit-downloads-and-user-contributions/18304-uvm-ea-ovm-compatibility-kit
  - Enhanced the compatibility layer to work with UVM 1.1 release
  - This layer sits on top of UVM and allows the migration to UVM w/o having to modify the OVM IPs

OVM Compatibility Layer

- The compatibility layer is done in a way that the existing OVM based environment can use it just as an OVM version change
  - The code below shows the ovm_pkg content which is derived from the UVM code base (green), the compatibility layer code are split into the files (red) as shown below

```
package ovm_pkg;
    `include "ovm_macros.svh"
     typedef class ovm_seq_item_pull_imp;
     typedef class ovm_seq_item_pull_port;
     `include "dpi/uvm_dpi.svh"
     `include "base/base.svh"
     `include "tlm1/uvm_tlm.svh"
     `include "comps/comps.svh"
     `include "seq/seq.svh"
     `include "tlm2/tlm2.svh"
     `include "reg/uvm_reg_model.svh"
     `include "compatibility/ovm_compatibility.svh"
     `include "compatibility/urm_message.sv"
     `include "compatibility/legacy_compatibility.svh"
endpackage
```

Mapping macros `uvm_* to `ovm_*

```
`include "uvm_macros.svh"
`define ovm_do_callbacks(CB,T,METHOD_CALL)
  `uvm_do_callbacks(T,CB,METHOD_CALL)
`define ovm_do_callbacks_exit_on(CB,T,METHOD_CALL,VAL)
  `uvm_do_callbacks_exit_on(T,CB,METHOD_CALL,VAL)
`define ovm_do_task_callbacks(CB,T,METHOD_CALL)
  `uvm_do_task_callbacks(T,CB,METHOD_CALL)
`define ovm_do_obj_callbacks(CB,T,OBJ,METHOD_CALL)
  `uvm_do_obj_callbacks(T,CB,OBJ,METHOD_CALL)
`define ovm_do_obj_callbacks_exit_on(CB,T,OBJ,METHOD_CALL,VAL)
  `uvm_do_callbacks(T,CB,METHOD_CALL)
`define ovm_do_obj_task_callbacks(CB,T,OBJ,METHOD_CALL)
  `uvm_do_obj_task_callbacks(T,CB,OBJ,METHOD_CALL)
`define ovm_do_ext_callbacks(CB,T,OBJ,METHOD_CALL)
  `uvm_do_ext_callbacks(T,CB,OBJ,METHOD_CALL)
`define ovm_do_ext_callbacks_exit_on(CB,T,OBJ,METHOD_CALL,VAL)
  `uvm_do_ext_callbacks_exit_on(T,CB,OBJ,METHOD_CALL,VAL)
`define ovm_do_ext_task_callbacks(CB,T,OBJ,METHOD_CALL)
  `uvm_do_ext_task_callbacks(T,CB,OBJ,METHOD_CALL)
`define ovm_cb_trace(OBJ,CB,OPER)
   `uvm_cb_trace(OBJ,CB,OPER)
```

Mapping classes uvm_* to ovm_*

```
// Typedefs: UVM->OVM
//
// Non-parameterized UVM classes can be simply typedefed to corresponding
// OVM types.
//------------------------------------------------------------------------------

typedef uvm_void                ovm_void;
typedef uvm_root                ovm_root;
typedef uvm_factory             ovm_factory;
typedef uvm_object              ovm_object;
typedef uvm_transaction         ovm_transaction;
typedef uvm_component           ovm_component;

// Parameterized UVM classes cannot be simply typedefed to corresponding
// OVM types, have to extend from uvm equivalents and pass the right parameters

class ovm_analysis_port #(type T=int) extends uvm_analysis_port#(T);
  function new(string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction
endclass
```

Mapping Enumerated Types

```
typedef uvm_active_passive_enum ovm_active_passive_enum;
uvm_active_passive_enum OVM_PASSIVE = UVM_PASSIVE;
uvm_active_passive_enum OVM_ACTIVE  = UVM_ACTIVE;

typedef uvm_verbosity ovm_verbosity;
parameter uvm_verbosity OVM_NONE   = UVM_NONE;
parameter uvm_verbosity OVM_LOW    = UVM_LOW;
parameter uvm_verbosity OVM_MEDIUM = UVM_MEDIUM;
parameter uvm_verbosity OVM_HIGH   = UVM_HIGH;
parameter uvm_verbosity OVM_FULL   = UVM_FULL;
parameter uvm_verbosity OVM_DEBUG  = UVM_DEBUG;

typedef uvm_severity ovm_severity;
uvm_severity OVM_INFO    = UVM_INFO;
uvm_severity OVM_WARNING = UVM_WARNING;
uvm_severity OVM_ERROR   = UVM_ERROR;
uvm_severity OVM_FATAL   = UVM_FATAL;
.....
```

UVM Source Code Change

- With the compatibility layer we can get majority of the OVM based VIPs and environments to compile clean
- But there were still few UVM files we have to change to make it 100% back-ward compatible to our OVM usage
  1. uvm_final/src/base/uvm_component.svh
  2. uvm_final/src/base/uvm_factory.svh
  3. uvm_final/src/base/uvm_globals.svh
  4. uvm_final/src/base/uvm_root.svh
  5. uvm_final/src/comps/uvm_driver.svh
  6. uvm_final/src/seq/uvm_sequencer.svh
  7. uvm_final/src/seq/uvm_sequencer_param_base.svh
  8. uvm_final/src/tlm1/sqr_connections.svh

uvm_component

- **Have to add pre_run() and call it from start_of_simulation phase**

```
function void uvm_component::start_of_simulation(); `ifdef OVM pre_run(); `endif
  return; endfunction
```

- **Have to add the `ovm_report_*` functions into the uvm_component**

```
`ifdef OVM
 function void uvm_component::ovm_report_info( string id,
                                                string message,
                                                int verbosity = UVM_MEDIUM,
                                                string filename = "",
                                                int line = 0);
   m_rh.report(UVM_INFO, get_full_name(), id, message, verbosity,
               filename, line, this);
 endfunction
`endif
```

uvm_factory

- **Have to add create_object() function into uvm_factory**

```
`ifdef OVM
static function uvm_object create_object (string requested_type_name,
                                          string parent_inst_path="",
                                          string name="");
   …
endfunction
`endif
```

- **Have to add set_inst_override function into uvm_factory**

```
`ifdef OVM
static function void set_inst_override (string full_inst_path,
                                        string original_type_name,
                                        string override_type_name);
   …
endfunction
`endif
```

uvm_globals and uvm_root

- Have to add ovm_test_top to uvm_globals.svh

```
`ifdef OVM
// This is set by run_test()
uvm_component ovm_test_top;
`endif
```

- Have to set ovm_test_top in uvm_root.svh

```
`ifdef OVM
    ovm_test_top = uvm_test_top;
`endif
```

Results

- **Successfully migrated a OVM based SoC to UVM using the OVM compatibility layer**
  - There were 25+ VIPs with complex bus interfaces like OCP/AHB/AXI and several I/Os like PCIE/USB/SDIO etc.
  - Have to add more code in the compatibility layer as few of the legacy IPs are even dependent on AVM compatibility layer in OVM
    - Ideally would be better to clean the IP source code to remove that legacy, but preferred to add support in compatibility layer as proof-of-concept
  - Managed to get all level-0 regressions containing 100+ tests passing
    - Took ~3 person weeks to enhance the compatibility layer
    - Took ~2 person weeks to run regressions and achieve same results as the reference
  - Filed several Mantis items on UVM source code based on the issues/bugs observed
    - e.g. print_topology() was crawling in UVM compared to OVM, simulator enhancements were needed to match OVM performance

Summary

- **Ideally, it would be nice to start from a clean code and not create a compatibility layer**
- **In our case this is not possible because of:**
  - The amount of OVM code that we have which needs to be converted and tested
  - The IPs that we get from all over the place internally and externally.
- **It will takes us years to use the clean code approach. Huge impact on**
  - Our resources
  - Execution schedule
- **Having the compatibility layer enables a SoC project to move to UVM when they decide and therefore without any effort or impact on execution schedule.**
- **This way a SoC can start developing new code in UVM and take opportunistic approach in converting old code as intercepts permit.**

A Reusable Verification Testbench Architecture Supporting C and UVM Mixed Tests

Richard Tseng

Qualcomm, Boulder CO

Agenda

- Introduction
- Testbench Features
- Testbench Architecture Overview
- Challenges & Solutions
- Summary
- Q&A

Introduction

Introduction

- **UVM would be the ultimate methodology for design verification**
  - It's the latest and greatest technology
  - Strong EDA vendor supports
  - VIPs and test sequences can be reusable
- **In many applications, C was primarily used for testing legacy designs**
  - Many scripts were written to support C tests
  - Thousand lines of C code have been evolved from many product generations
  - They are recognized as "golden regression suite"
- **The goal is to build an UVM testbench**
  - To re-use verification components
  - To re-use existing C and UVM test sequences

Testbench Features

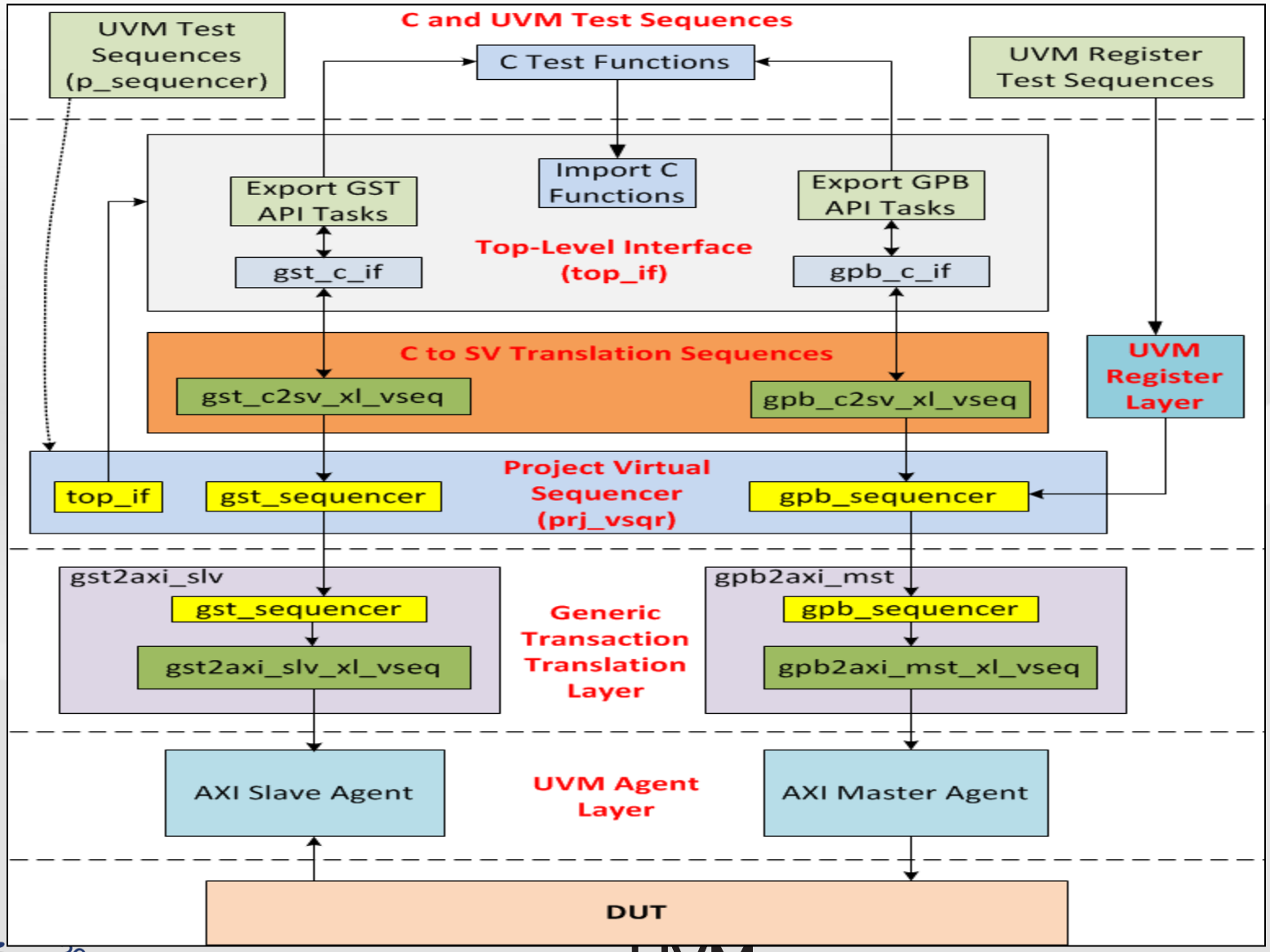
Testbench Features

- **The UVM testbench architecture allows us:**
  - To reuse C and UVM tests in various platforms
  - To run C/UVM tests simultaneously
  - To reuse UVM verification components
  - To easily integrate UVM register layer

Testbench Architecture

Overview

Testbench Architecture Overview



Challenges & Solutions

Challenges & Solutions

1. Creating UVM sequences with API Tasks
2. Reusing high-level C and UVM tests and testbench components
3. Integrating UVM Register Layer

Challenges & Solutions

1. **Creating UVM sequences with API Tasks**
2. Reusing high-level C and UVM tests and testbench components
3. Integrating UVM Register Layer

Creating UVM sequences with API tasks

- **The C tests are usually programmed with API tasks**
- **Typically in an UVM SVTB, UVM macros are used to create a test sequence,**
  - ie, `uvm_do(),`uvm_do_with(), and `uvm_do_on_with()
- **UVM "do" macros don't match the legacy C tests, using the API task is more consistent**
- **To reuse the C tests in an UVM testbench, C API tasks need to be mapped to UVM "do" macros**
  - Constraints can be specified in API task arguments
  - Default constraints can be specified within the API task

UVM “do” macro V.S. API task

- **Transaction generated with “do” macro:**

```
`uvm_do_with(req, axi_mst_sqr,{req.addr == 'h1000;
                               req_data == 'h1234_5678;
                               req.cmd_type == AXI_WR_INC;
                               req.burst_length == 1;
                               req.burst_size == 4;
                               req.bready_delay == 1;
                               req.avalid_delay == 0;
                               …})
```

- Equivalent API task being called:

```
// axi_master_write(address, data)
axi_master_write('h1000, 'h1234_5678);
```

Create Bus Transaction with API task

```
task axi_master_base_seq::axi_mst_write(input bit [31:0] addr,
                                        input bit [31:0] data);

  `uvm_create(req)

  assert (req.randomize() with {
      req.cmd_type     == AXI_WR_INC;
      req.address      == addr;
      req.data         == data
      req.burst_length == 1;
      req.burst_size   == 4;
      req.bready_delay == 1;
      req.avalid_delay == 0;
    }) else begin
      `uvm_fatal(…..)
    end

  `uvm_send(req)

endtask: axi_mst_write
```

Create a sequence item

Randomize with address, data, and default constraints

Send to sequencer

Challenges & Solutions

1. Creating UVM sequences with API Tasks
2. **Reusing high-level C and UVM tests and testbench components**
3. Integrating UVM Register Layer

Map generic transactions to bus interface specific API tasks

- **Portable tests are composed of generic API tasks**
- **GPB (Generic Primary Bus) tasks**
  - Mapped to front-door register interface/primary interface transactions:
    - gpb_write() => axi_master_write()
    - gpb_read() => axi_master_read()
- **GST (Generic Slave Transaction) tasks**
  - Mapped to slave device's back-door transactions
    - gst_bkdr_write() => axi_slave_bkdr_write()
    - gst_bkdr_read() => axi_slave_bkdr_read()

Map generic transactions to bus interface specific API tasks (cont'ed)

```
// generic test sequence
task my_test_seq::body():
  gpb_write(reg1, data);
  gpb_read(reg2, data);
  gpb_nop(10); // idle for 10 clocks
  gst_bkdr_write(32'h0, 32'h1234_5678);
  gst_bkdr_read(32'h5555_aaaa, read_data);
endtask
```

```
//bus interface specific sequence
task my_test_seq::body():
  axi_master_write(reg1, data);
  axi_master_read(reg2, data);
  axi_master_nop(10); // idle for 10 clocks
  axi_slave_bkdr_write(32'h0, 32'h1234_5678);
  axi_slave_bkdr_read(32'h5555_aaaa, read_data);
endtask
```

Make C and UVM tests identical!

```
class sv_main_seq extends prj_base_seq;
  task body();
      bit[31:0] read_data;
      `uvm_info(get_type_name(), “Test starts”, UVM_MEDIUM)
      gpb_write(control_reg, 32’h0000_3204);
      gpb_read(status_reg, read_data);
      gst_bkdr_write(‘h400, 32’hA5);
      gst_bkdr_read(‘h400, read_data);
  endtask: body
endclass: sv_main_seq
```

Same UVM macro

```
void c_main_seq(void) {
  unsigned int read_data; // 32 bit unsigned integer
  uvm_info(__func__, “Test starts”, UVM_MEDIUM)
  gpb_write(control_reg, 0x00003204);
  gpb_read(status_reg, &read_data);
  gst_bkdr_write(0x400, 0x000000A5);
  gst_bkdr_read(0x400, &read_data);
 …
}
```

Using UVM reporting macros in C

- **UVM has a good reporting service**
  - Can specify the message verbosity level and severity
  - Generate messages which show where and when they are called during simulation

- **Macros include `uvm_info(), `uvm_warning(), `uvm_error(), `uvm_fatal()**

Implement UVM reporting macros in C

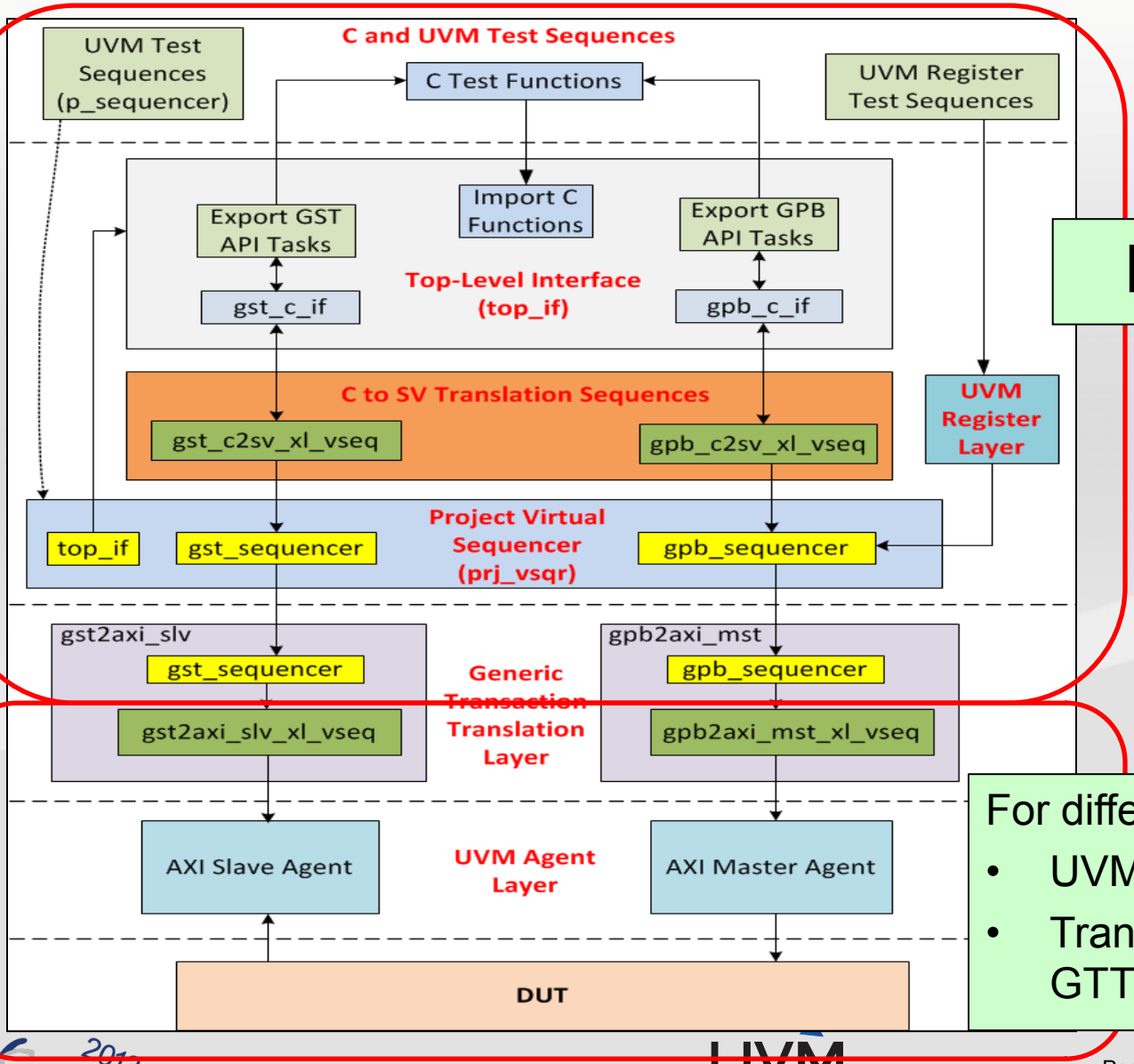
```
// In SV file, define uvm_rpt_info() function
function uvm_rpt_info(string id, string message,
                      int verbosity = UVM_MEDIUM);
  `uvm_info(id, message, verbosity)
endfunction: uvm_rpt_info
```

```
// export uvm_rpt_info function
export "DPI-C" function uvm_rpt_info;
```

```
// In C file, define verbosity level just as UVM definitions
#define UVM_LOW    100
#define UVM_MEDIUM 200
#define UVM_HIGH   300
```

```
// define C macros
#define uvm_info(id, message, verbosity) \
    uvm_rpt_info(id, message, verbosity);
```

Reuse high-level VIPs



Reusable!

For different deign, replace:

- UVM Agent Layer
- Translation sequences in GTTL

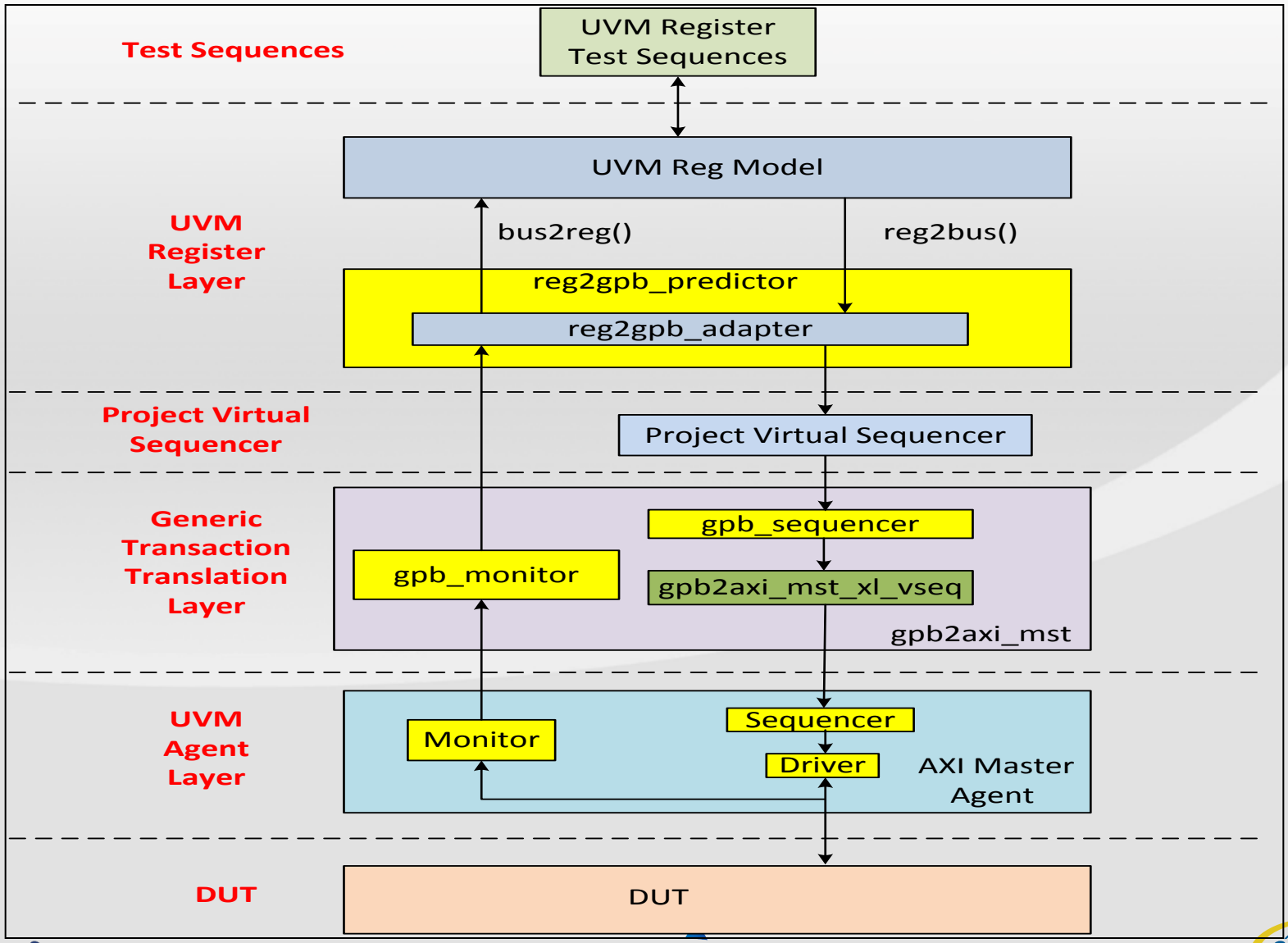
Challenges & Solutions

1. Creating UVM sequences with API Tasks
2. Reusing high-level C and UVM tests and testbench components
3. **Integrating UVM Register Layer**

UVM Register Layer integration

- **Register models are auto generated by customized scripts or EDA vendor provided tools**
- **Explicit prediction mechanism is used in our example**
  - It's recommended prediction mechanism
  - Register model will be updated
    - Register sequences issued from register sequencer (auto prediction)
    - Bus transactions issued from all other bus agents (passive prediction)
- **Required additional verification components to be added, so that the followings can be reused across different platforms:**
  - **Register sequences**
  - **Predictor**

UVM Register Layer Integration



Summary

Summary

- **In our applications, test sequences and the VIPs were reused in multiple testbenches**
  - Modem core–level testbench
    - GPB => QSB AXI master
    - GST => QSB AXI salve
  - Modem block-level testbenches
    - GPB, GST => proprietary bus interfaces
  - Modem emulation platform testbench
    - GPB => AHB Master
    - GST => Off-chip ZBT memory
  - Regressions have been run with multiple simulators
- **The testbench architecture extends the reusability beyond the scope of the UVM technology, and across the C and SV language boundary**

Q&A

UVM to the Rescue – Path to Robust Verification

Asad Khan

Design Verification Lead

Texas Instruments, Inc.

Agenda

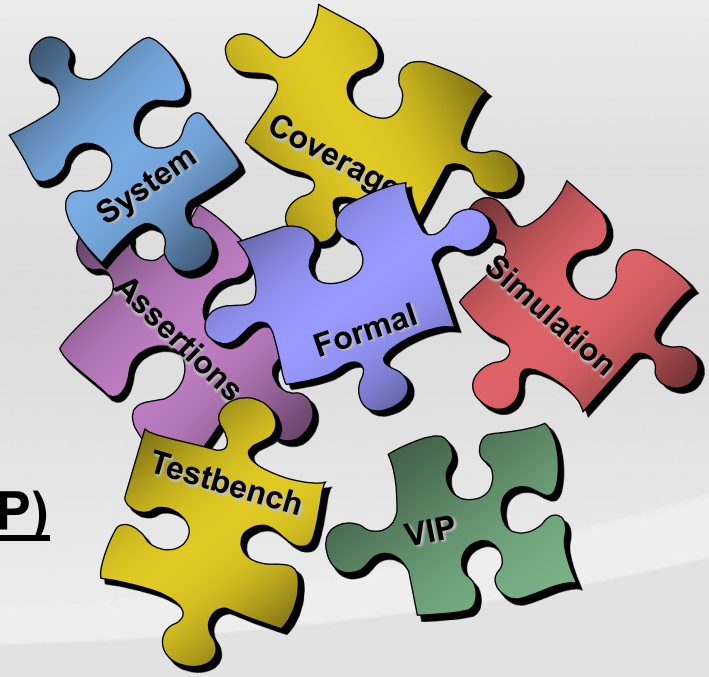
1. Our Challenges Before…
2. Adopted Solutions and Roadblocks
3. UVM to the Rescue – Key Issues Addressed
4. Issues Detailed & UVM Solution
5. UVM in AMS Simulation
6. Conclusions
7. Q/A

Our Challenges Before…

Lack of constrained random stimulus

Looks of several verification environments patched together

Lack of functional coverage driven methodology



Lack automated checking aspects

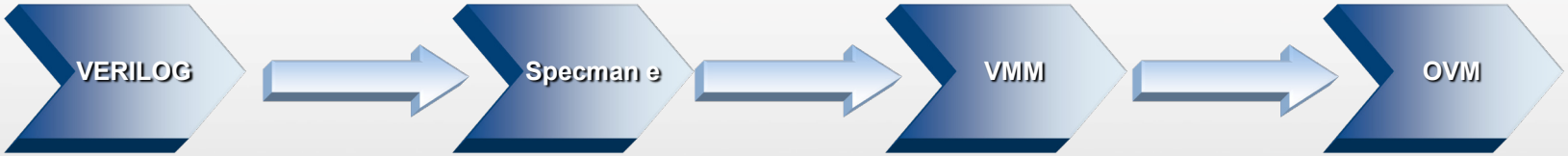
Lack structured methodology and aspects of reusability

Lack of provisioning for verification IP (VIP) support

Lack of block level to top level test back compatibility

Bulky in nature with directed test overload – a management nightmare

Adopted Solutions and Roadblocks



Language Specific – Needing Expertise

Lack of Full LRM Support – Not all features supported

Not Open Source – Requiring Support, Licenses and Customizations

Simulator Dependent – Compatibility issues with Major Options

Lack of Base Class Functions – A lot of coding e.g. deep copy etc

Needed Wrapper Development to build a Test Environment

Lack of Phases and Several Roadblocks in Methodology

VIP Solutions Lacking Standardization – Everyone Seemed to have their own methodology

UVM to the Rescue – Key Issues Addressed

Packet Class Issues	Issues due to rewrite of the entire task/function for enhancing it for the newly added data member of the packet class – no automation
End of Test Issues	Waiting for pending items & drain time/delays caused end of test issues
Inconsistency in Test Development	Issues e.g. errors, directed test overload due to lack of base test class
Stimulus Control Issues	Controlling multiple DUT Interfaces through Verification Components resulted in complex and non reusable scheme due to Callbacks
BFM/Monitor Reuse Issues	Lack of schemes to reuse BFMs/Monitors from legacy test environments
Lack of Debug Messaging Support	Lack of or Non reusable custom environment printers during runtime caused environment debug issues including complex simulation logs
Miscellaneous Testbench Issues	Issues in writing more tasks to process channel data , repetitive and non-reusable, lack of phasing options, and extensive coding of register models

Packet Class Issues

From Issue to Solution

We have faced issues in rewriting functions/tasks to override/cast e.g. for copy, display and all others, however, using UVM object_utils the data item automation is built in with field level flags

Example: Copy Function Issue

```
class ahb_master_trans extends vmm_data;
  //DATA members
  rand integer unsigned NumBytes = 0;
  rand integer unsigned NumBeats = 1;
  rand integer busy;
  rand byte data [$];
  rand bit [(`AHB_HADDR_WIDTH -1):0] address = 'h0;
  …

  function vmm_data copy(vmm_data to = null);
    ahb_master_trans cpy;
    super.copy_data(cpy);
    cpy.NumBytes      = this.NumBytes;
    cpy.NumBeats      = this.NumBeats;
    cpy.busy          = this.busy;
    cpy.address       = this.address;

    for(int i = 0; i<this.NumBytes ; i++) begin
      cpy.data[i] = this.data[i];
    end
    copy = cpy;
  endfunction:copy
endclass
```

UVM: Built-in Automation

```
class ahb_master_trans extends uvm_transaction;
  //DATA members
  rand integer unsigned NumBytes = 0;
  rand integer unsigned NumBeats = 1;
  rand integer busy;
  rand byte data [$];
  rand bit [(`AHB_HADDR_WIDTH -1):0] address = 'h0;
  …

 `uvm_object_utils_begin(ahb_transfer)
    `uvm_field_int(NumBytes , UVM_ALL)
    `uvm_field_int(NumBeats , UVM_ALL)
    //Similarly for other fields
 …….
 …….
 `uvm_object_utils_end
 ….
endclass
```

UVM_NOCOMPARE, UVM_NOPRINT etc

Data Item Automation in Place!!

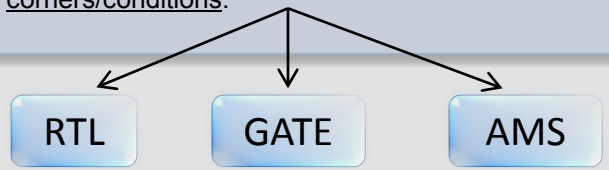
End-of-Test Issues

From Issue to Solution

We have always struggled with the issue on how to achieve "end of test". Logically when all activity completes should be the end of test, however, checkers could still be checking or testbench components can be busy - how do we reach true end-of-test?

Problematic End-of-Test Approaches

- In cases where simulation had to be stopped based on multiple interacting components, custom approaches were used e.g. detecting occurrence of a condition and using events – This was complex, cumbersome and not reusable.
- Callbacks were used in VMM for specific end of test conditions such as monitor observing pass/fail condition. Several times debug time had to be put in when the callback was missed from vmm_env::build(), and this approach made env non-reusable since test conditions changed per test scenarios.
- A crude "drain time" implementation was used to terminate tests after a certain delay but this resulted in different end of test times for sims at different corners/conditions.



UVM: uvm_test_done

- Using several techniques in UVM we were able to achieve uniform and highly reusable end of test solution:
  - Inside tests, sequences/virtual sequences & uvm components:
    - uvm_test_done.raise_objection (this);
    - uvm_test_done.drop_objection (this);
  - Inside particular phase
    - phase.raise_objection(this);
    - phase.drop_objection(this);
  - Inside run_phase
    - global_stop_request()
  - For debug and status of objections:
    - phase.phase_done.display_objections();

Inconsistency in Test Development

From Issue to Solution

Tests created as program blocks always resulted in inconsistent DUT initializations, redundant code with a test becoming a testbench, and in some cases incorrect sequences because of multiple test writers using independent approach – Test Maintenance Nightmare!

program block test issues

```
program my_test(myInterface.TxRx  my_host_if);
  `include “my_vmm_include.sv"
   vmm_log  log = new(“My Env", "main-class");
   my_env env;
   ……
   initial begin
     // I don’t know but I am configuring wrong
     dut_register_base = 12’hBAD;
     // I am sending good packets to wrong place
     send_link_pkt();
     while (state!==0) begin  // Wait For Complete
        read32(dut_register_base), tmp_data[31:0]);
        state = tmp_data[11:8];
     end
     $display(“Device has reached U0 successfully\n");
     $display("Test Passed \n");
   end
endprogram
```

False Pass since “state” for this bad address is non-zero by default where expectation was that after sending link packet the correct address would have become non zero

UVM: uvm_test Class

```
class test_base extends uvm_test;
   top_env env;
   `uvm_component_utils(test_base)
  ….
  virtual function void end_of_elaboration_phase(uvm_phase phase);
      dut_register_base = 12’hAAA;
  endfunction:end_of_elaboration_phase
  …..
endclass


class my_test extends test_base;
   `uvm_component_utils(my_test)
   ….
   task run_phase(uvm_phase phase);
     send_link_pkt();
     while (state!==0) begin  // Wait For Complete
        read32(dut_register_base), tmp_data[31:0]);
        state = tmp_data[11:8];
     end
     $display(“Device has reached U0 successfully\n");
     $display("Test Passed \n");
 endtask:run_phase
endclass
```

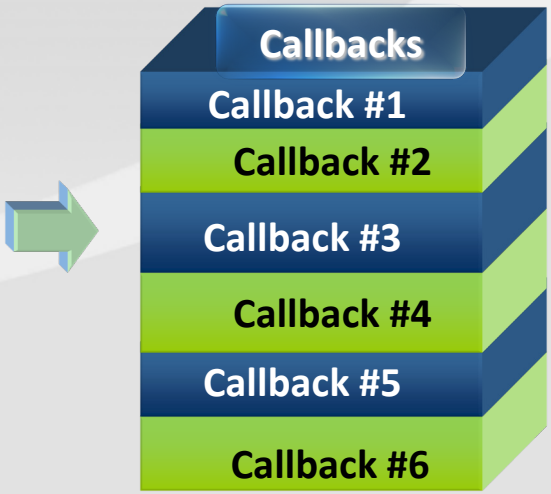
Stimulus Control Issues

Issue at Hand?

DUT with multiple interfaces needed interactive and interdependent control of stimulus and response to incorporate feedback from the simulation to direct the transaction generation. This was implemented using Callbacks and became a critical issue from test to test.

Verification Challenge

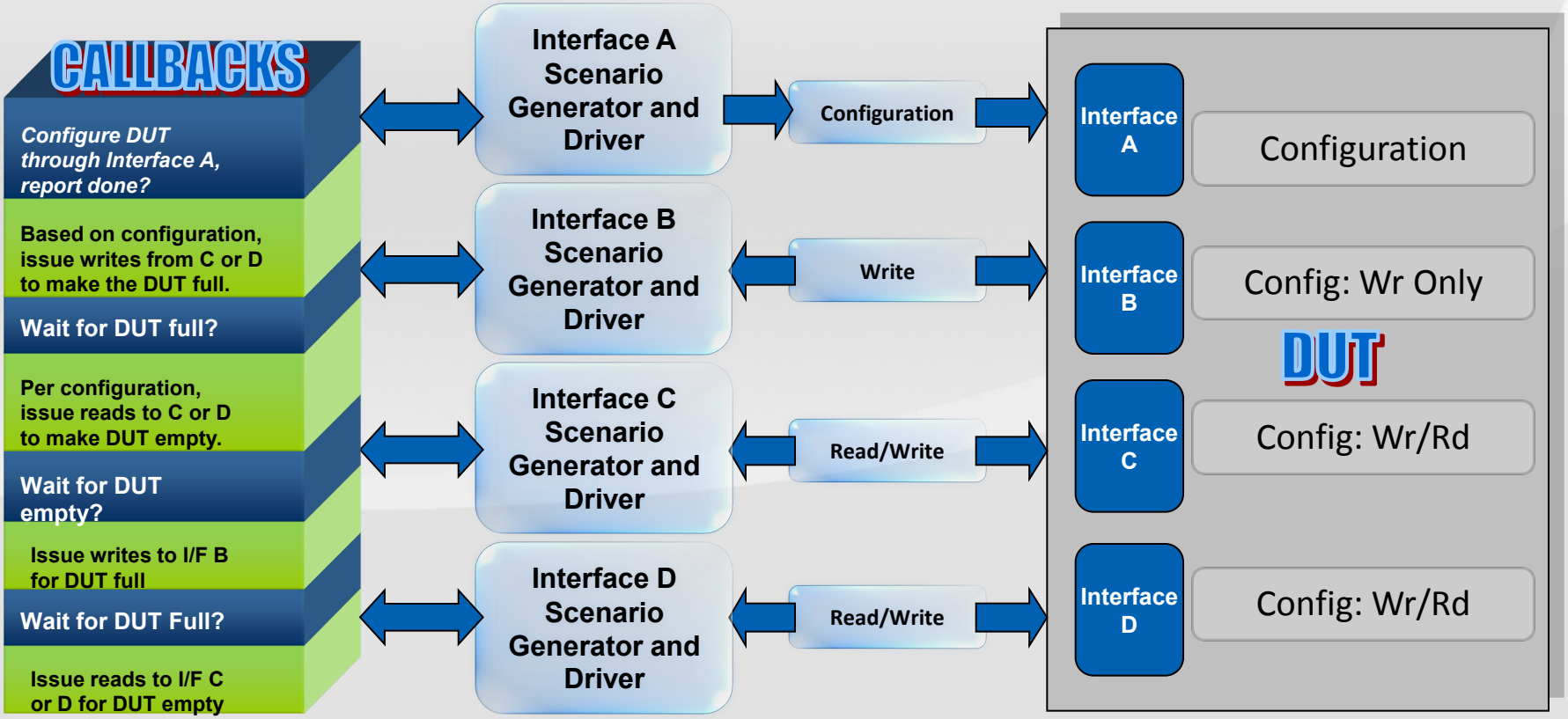
- *Four interfaces A, B, C and D that need to be exercised.*
- *Interface A configures whereas B, C and D can either be used to read or write.*
- *Interfaces B, C and D have to wait until configuration is complete.*
- *Testing full and empty conditions involve coordination between individual interfaces.*
- *Reconfiguration on A can happen anytime making B, C and D read or write or vice versa.*



Stimulus Control Issues (Cont.)

Example Case 1

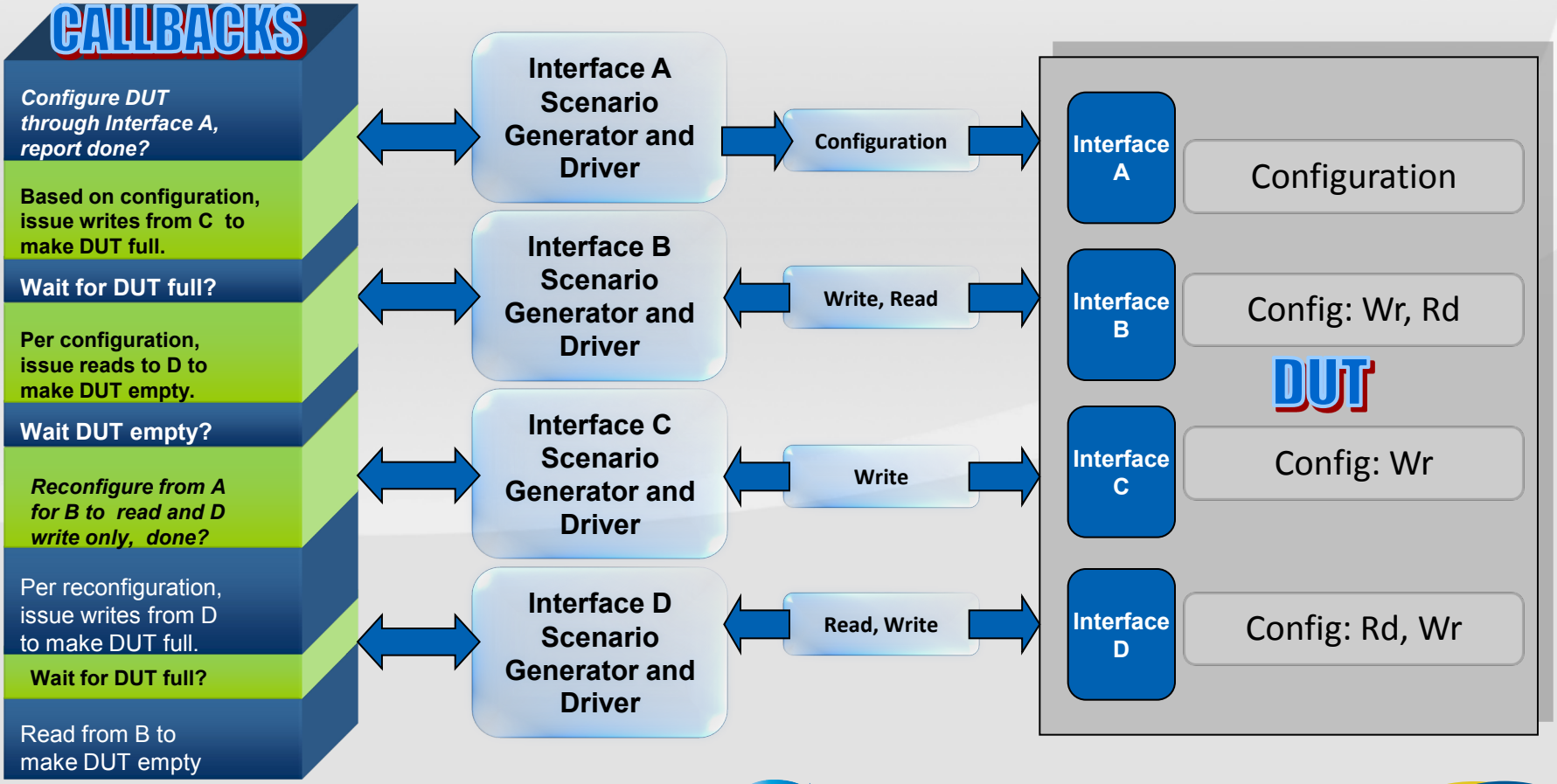
- Configure DUT using interface A so interface C and D can both Read & Write
- Configure Interface B to be Write only
- Using interface B, C and D make two iterations of DUT Full to Empty



Stimulus Control Issues (Cont.)

Example Case 2

- Configure DUT through interface A so that interface C is write, D is read, and B is write only
- Reconfigure from A to make B read and D write only



Stimulus Control Issues (Cont.)

From Issue to Solution

Using virtual sequences and nested sequences in UVM we were able to create a randomized approach to configure interface B, C and D as either read, write or both and also do full and empty sequencing.

Example Code Snippet of Issue

```
program dut_test(..);    //beginning of testcase
  //callback for controlling and generating transfers
  class write_read extends int_master_callback;
    virtual task write(master_trans transaction = null);
    begin
      if(transaction.int == `INTF_A) begin
        //wait for the configuration write to complete
      end else begin
      if(transaction.int == `INTF_B && env.DIR == 1'b0)
        env.chan.put(wr_trans);  end
    endtask
    virtual task read(master_trans transaction = null);
      env.intf_cl_chan.put(rd_trans); endtask
  endclass
initial begin
    env.build();
    env.intf_a_xactor.append_callback(w0_full_clbk);
    env.intf_c_xactor.append_callback(w0_full_clbk);
    env.intf_d_xactor.append_callback(w0_full_clbk);
    env.run();

end endprogram      //end of testcase
```

UVM: Nested and Virtual Sequences

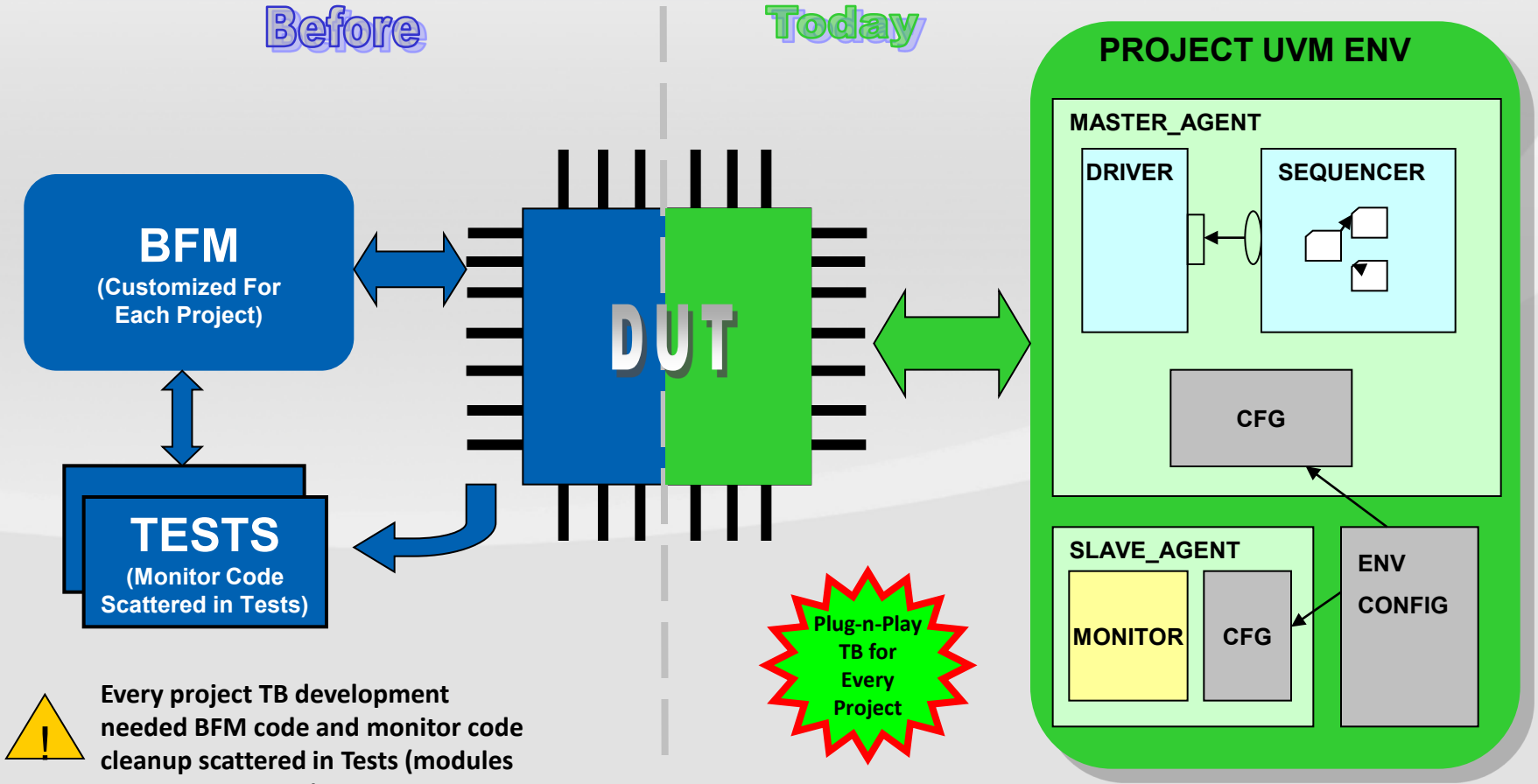
```
class intf_sequence extends virtual_sequence_base;
…..
 `uvm_object_utils(intf_sequence)
  ….
  virtual task body();
   super.body();
   …
   `uvm_do(cfg_init) // Configure using intf A per cfg_init_intf_*
   for(int i = 1;i<=iterations;i++)
    `uvm_do_on_with(intf_seq, intf_sequencer, {
                    recfg  == `INT.reconfigure;
                    tr1init == `INT.tr1init_order;
      ….})
  endtask
endclass

constraint int_config::test_specific {
  reconfigure    == TRUE;          iterations       == 1;
  tr1_init_order  == C_FULL;       tr1_final_order  == C_EMPTY;
  tr2_init_order  == D_EMPTY;      tr2_final_order  == D_FULL;
  tr3_init_order  == B_FULL;       tr3_final_order  == B_EMPTY;
  cfg_init_intf_B == RD_OR_WR;  cfg_final_intf_B == RD_ONLY;
  cfg_init_intf_C == RD_OR_WR;  cfg_final_intf_C == RD_ONLY;
  cfg_init_intf_D == RD_ONLY;    cfg_final_intf_D == WR_ONLY; }
```

BFM/Monitor Reuse Issues

From Issue to Solution

Our BFM/Monitor reuse from project to project has always resulted in to a non-structured code with several redundancies. This always involved re-structuring and recode on every project – time wasted!



Every project TB development needed BFM code and monitor code cleanup scattered in Tests (modules or program blocks).

Lack of Debug Messaging Support

From Issue to Solution

We have always faced issues because our test environment lacked debug friendly features. Additional code had to be written to make logs meaningful, print more info on the test environment components and configurations.

Legacy TB Debug Support

Case-1:

```
if (condition_happens)
   $display("Print Details");

   `vmm_note(this.log,$psprintf("ADDR: %h",address));
```

Case-2:

```
`define WARNING $write("WRN: %t %m",$time); $display
`define ERROR   $write("ERR: %t %m",$time);  $display
`define DEBUG
        if(debug) $write("DBG: %t %m",$time);
        if(debug) $display
```

Case-3: Compile with +DEBUG

```
`ifdef DEBUG
   $display("Print Details");
 if($test$plusargs("DEBUG")) debug = 1;
   `vmm_fatal(this.log,"Data Mismatch");
        `WARNING("Issue a Warning");
        `ERROR("Issue an Error");
        `DEBUG("Debug Debug Debug");
```

UVM: Built-in Messaging

```
function void end_of_elaboration_phase(uvm_phase phase);
   env.assertions_inst.set_report_verbosity_level(UVM_MEDIUM);
   env.scbd.set_report_verbosity_level(UVM_LOW);
   env.monitor.set_report_verbosity_level(UVM_HIGH);
endfunction : end_of_elaboration_phase
```

```
`uvm_info(get_type_name(),$psprintf("%d",val), UVM_LOW);
`uvm_info(get_type_name(),$sformatf("%d",$time),UVM_NONE)
`uvm_info("INFO1","Print This..",UVM_HIGH)
`uvm_info("INFO2","More Print..",UVM_MEDIUM)
```

Compile with +UVM_VERBOSITY=**UVM_FULL | UVM_LOW | UVM_MEDIUM**

```
    `uvm_warning("WARNING", "Issue a Warning")
    `uvm_error("ERROR", string message)
    `uvm_fatal(string id, string message)
```

Lack of Debug Messaging Support (Cont.)

Hierarchy Print & Report Server

Using UVM messaging support we were able to print the TB hierarchy and also customize the report server per our requirements.

```
uvm_report_server my_rpt;
virtual function void report_phase(uvm_phase phase);
    int error_cnt, fatal_cnt, warning_cnt;
    my_rpt = _global_reporter.get_report_server();
    error_cnt = my_rpt.get_severity_count(UVM_ERROR);
    fatal_cnt = my_rpt.get_severity_count(UVM_FATAL);
    warning_cnt = my_rpt.get_severity_count(UVM_WARNING);
    if(error_cnt != 0 || fatal_cnt != 0) begin
      `uvm_info(get_type_name(), "\n SIM FAILED \n", UVM_NONE);
    end else begin
      `uvm_info(get_type_name(), "\n SIM PASSED \n",UVM_NONE);
    end
endfunction: report_phase
```

Customizing report_server

```
class custom_report_server extends uvm_report_server;
  virtual function string compose_message( uvm_severity severity,
    string name, string id, string message, string filename, int line );
    uvm_severity_type severity_type = uvm_severity_type'( severity );
    if(severity_type == UVM_INFO)
      return $psprintf( "%0t | %s | %s", $time, id, message );
    else
      return $psprintf( "%s | %0t | %s | %s | %s",
          severity_type.name(), $time, name, id, message );
  endfunction: compose_message
endclass: custom_report_server

function void start_of_simulation();
    custom_report_server custom_server = new;
    uvm_report_server::set_server( custom_server );
endfunction: start_of_simulation
```

```
task run_phase(uvm_phase phase);
  printer.knobs.depth = 5;
  `uvm_info("ENV_TOP",
            $psprintf("\n\n\nPrinting the test topology:"), UVM_HIGH)
  `uvm_info("ENV_TOP",
            $psprintf("Printing...\n\n %s",this.sprint(printer)), UVM_HIGH)
endtask: run_phase
```

Global Control on TB Hierarchy printing

```
Printing the test topology:
UVM_INFO @ 0 ns: uvm_test_top.env [ENV_TOP] Printing...
--------------------------------------------------------------------------------
Name                            Type                        Size  Value
--------------------------------------------------------------------------------
env                             top_env                     -     @1195
  my_i2c_util_inst              my_i2c_util                 -     @9017
  my_rgm_rdb_inst               my_register_db              -     @7634
  my_address_map                my_address_map_type         -     @9532
  my_user_assertions_inst       my_user_assertions          -     @9227
    env_dsi_cfg                 tb_cfg_chan_a               -     @7800
      master_config             agent_config                -     @7791
        parent_component        agent                       -     @9303
          driver                driver                      -     @12785
          monitor               my_monitor                  -     @12085
          sequencer             seq_sequencer               -     @12092
          cfg                   agent_config                -     @7791
          recording_detail      uvm_verbosity               32    UVM_FULL
--------------------------------------------------------------------------------
```

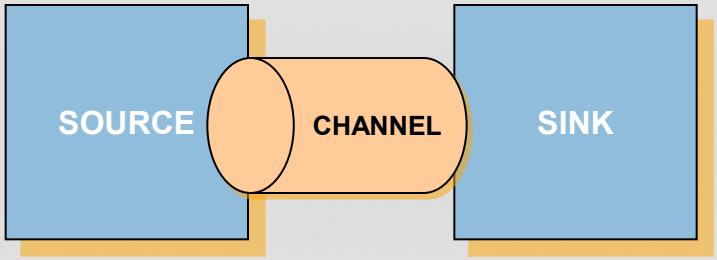
Miscellaneous Testbench Issues

Issue at Hand?

We saw several problems in implementation using data channels where additional code had to be written for data handling, and also in cases where various simulation phases had to be handled.

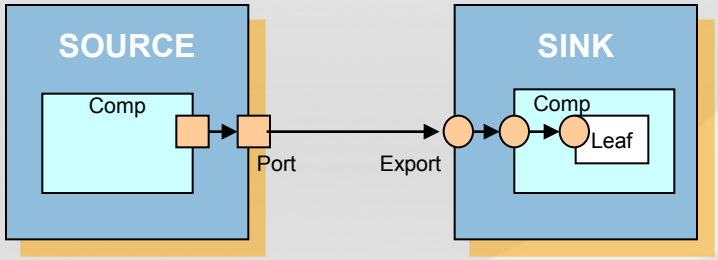
Data Transactions Between Blocks

BEFORE



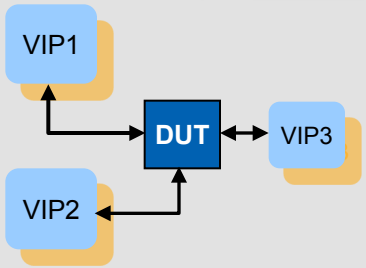
Needed to write tasks for data processing from vmm_channel

AFTER



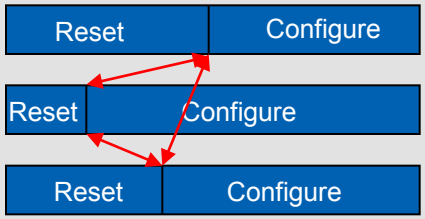
TLMs were reusable, available functions, and simplified implementation

Synchronizing Verification Components



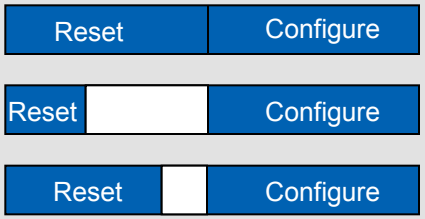
BEFORE

Events were used to synch and control



AFTER

Phasing support in UVM supported synchronization



Miscellaneous Testbench Issues (Cont.)

Issue to Solution

We were always <u>manually writing our configuration register models</u> that was <u>time consuming and full of errors</u>. Using SPIRIT (IPXACT) scripts we were able to <u>automate our register model code</u> generation for UVM.

BEFORE: A LOT OF MANUAL CODING!!!

AFTER

	B	C	D	E	F	G	H	I	J	K
1	DESCRIPTION	ADDRESSOFFSET	SIZE	ACCESS	RESET(VALUE)	RESET(MASK)	FIELDNAME	FIELDOFFSET	FIELDWIDTH	FIELDACCESS
2		0x0	8	RW	0x35	0xFF	DEVICE_ID	0	8	RO
3		0x1	8	RW	0x38	0xFF	DEVICE_ID	0	8	RO
4		0x2	8	RW	0x49	0xFF	DEVICE_ID	0	8	RO

perl gen_spirit_from_excel.pl project_reg_map.xls

java -jar $UVM_RGM_HOME/builder/ipxact/uvmrgm_ipxact2sv_parser.jar -input project_reg_map.spirit -ov -ve internal

```
Generating output file with default settings
Input file: project_rgm.spirit
Output file: project_rgm.sv
Logfile: ipxact2sv.log
Vendor Extensions usage: 'internal vendor extensions'
Class Definition postfix: _type
Parsing input file...
Normalizing XML file tags...
Generating java model from input file...
Writing SystemVerilog file...
Number of registers extracted = 988
Number of regFiles extracted = 1
Number of addrMaps extracted = 1
Parser is exiting...
Output file has been generated at: project_rgm.sv
___Execution complete___
```

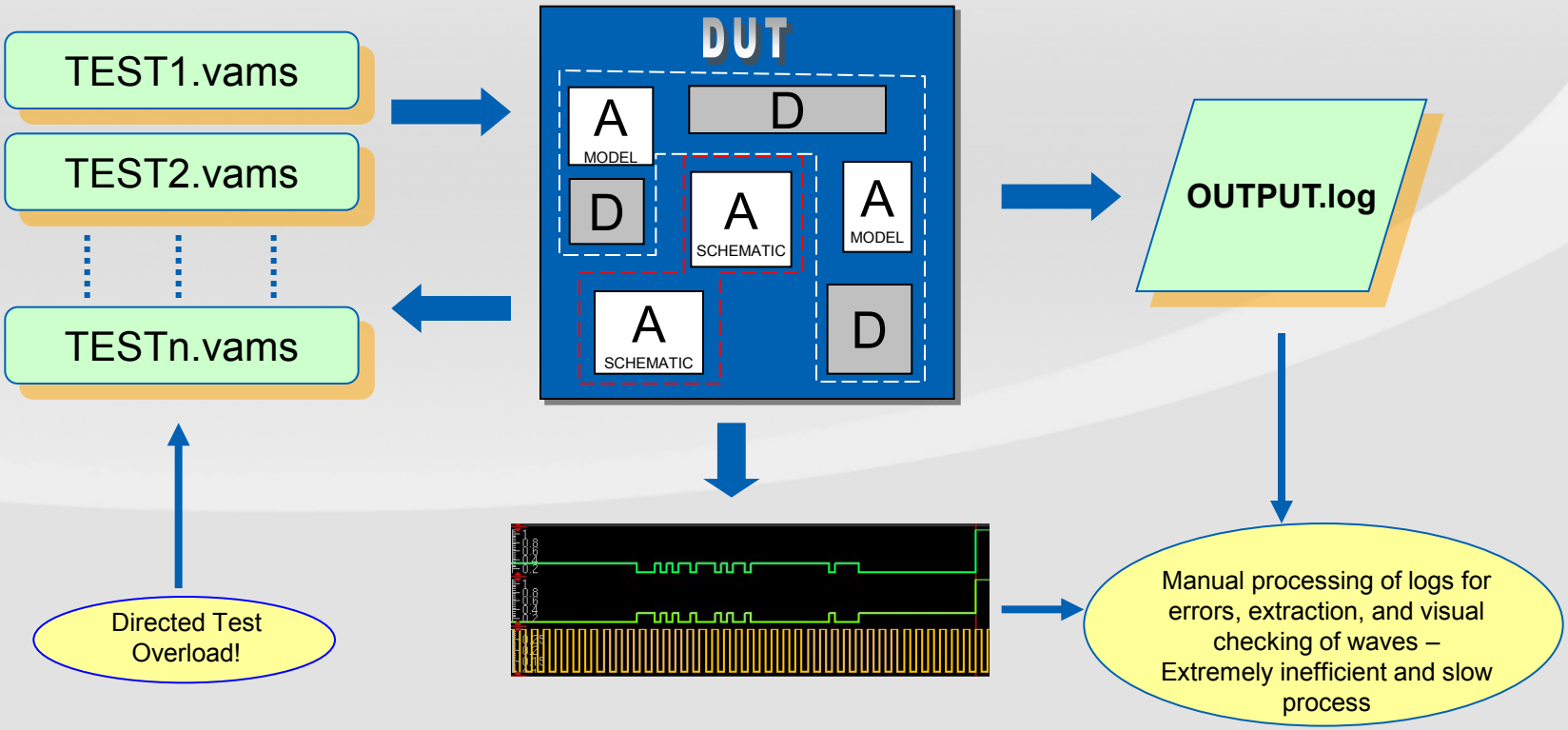
```
class DEVICE_ID0_type extends uvm_rgm_sized_register #(8);
  typedef struct packed {
    logic [7:0] DEVICE_ID0;
  } pkd_flds_s;
  `uvm_object_utils(DEVICE_ID0_type)
  ........
  ........
endclass : DEVICE_ID0_type

class project_register_db extends uvm_rgm_db;
  rand project_address_map_type project_address_map;
  `uvm_component_utils(project_register_db)
  .....
  virtual function void build();
    super.build();
    // Create the address map
    project_address_map = project_address_map_type::type_id::create("project_address_map", this);
    add_addr_map(project_address_map);
  endfunction : build
endclass : project_register_db
```

UVM Usage in AMS Simulations

Before

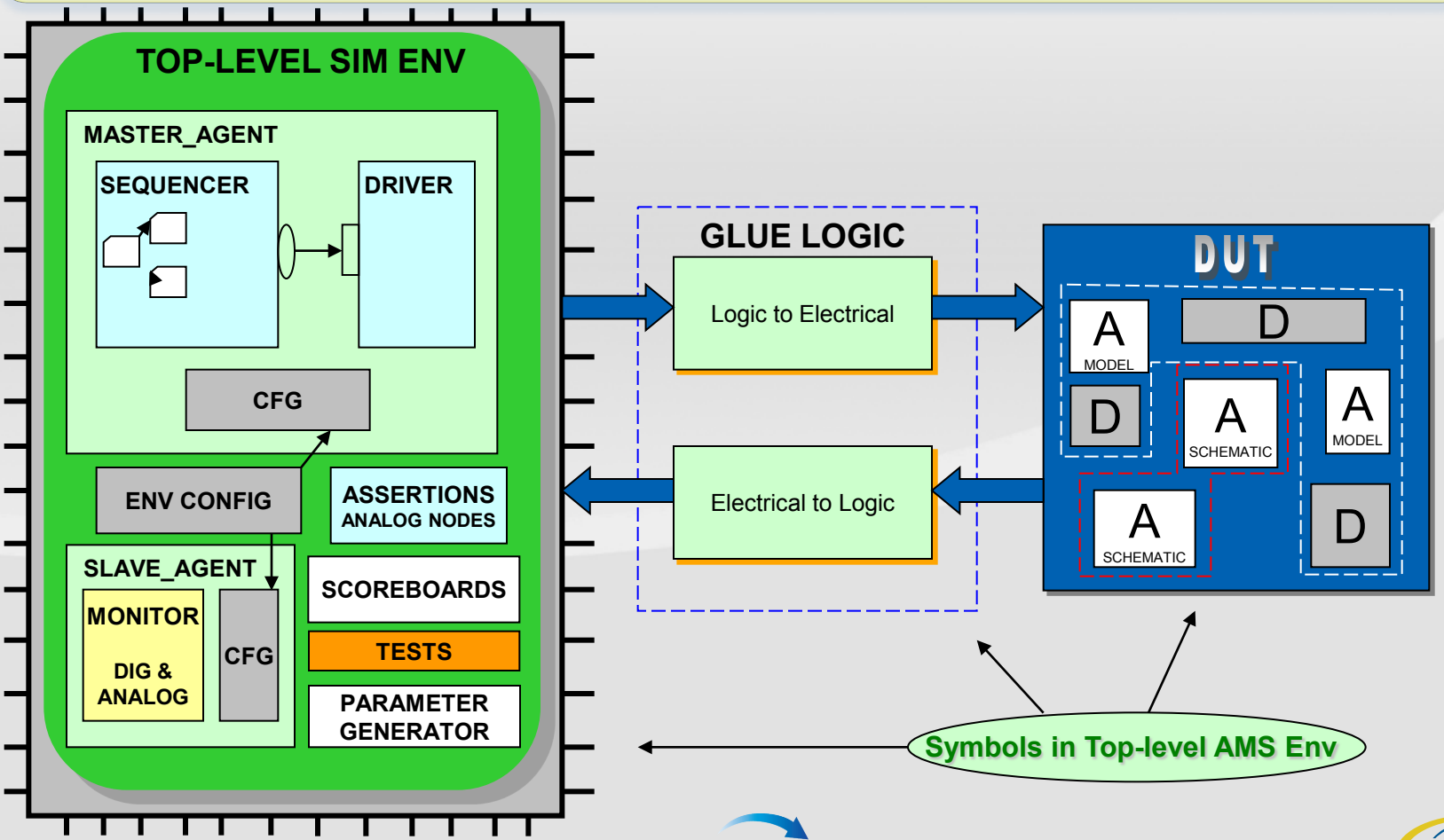
We faced problems using constrained-random top-level testbench in AMS environment because of compile problems and inconsistent methodologies, and had to rely on non-standard and non-reusable verification.



UVM Usage in AMS Simulations (Cont.)

After

Using UVM infrastructure we were able to re-use the same testbench and test suite that was created for RTL/GATE level also for AMS simulation allowing us to run an end-to-end simulation with packet traffic and protocol.



Technical Contributors

- Paul Howard
- Ravi Makam
- Jim Skidmore
- Chuck Branch
- Shyam Narayan
- Arun Mohan
- Pradeep Hanumansetty
- Ronnie Koh

Conclusions

- **UVM cleanly addressed our critical issues that were causing significant slowdown and down time due to code re-write**
- **UVM development goals align with our verification strategy/roadmap**
- **We did see some conversion effort in going from UVMEA1.0 to UVM1.1 but this effort was minimal**
- **We found UVM helpful in following ways:**
  - Getting started with UVM was easy – lots of trainings and guidance
  - We were able to develop complex test environments quickly
  - We found that available VIPs following UVM make integration and usability easier
- **We are today using UVM actively in our Digital and Mixed signal verification, and plan to use in Analog verification also**

Q/A

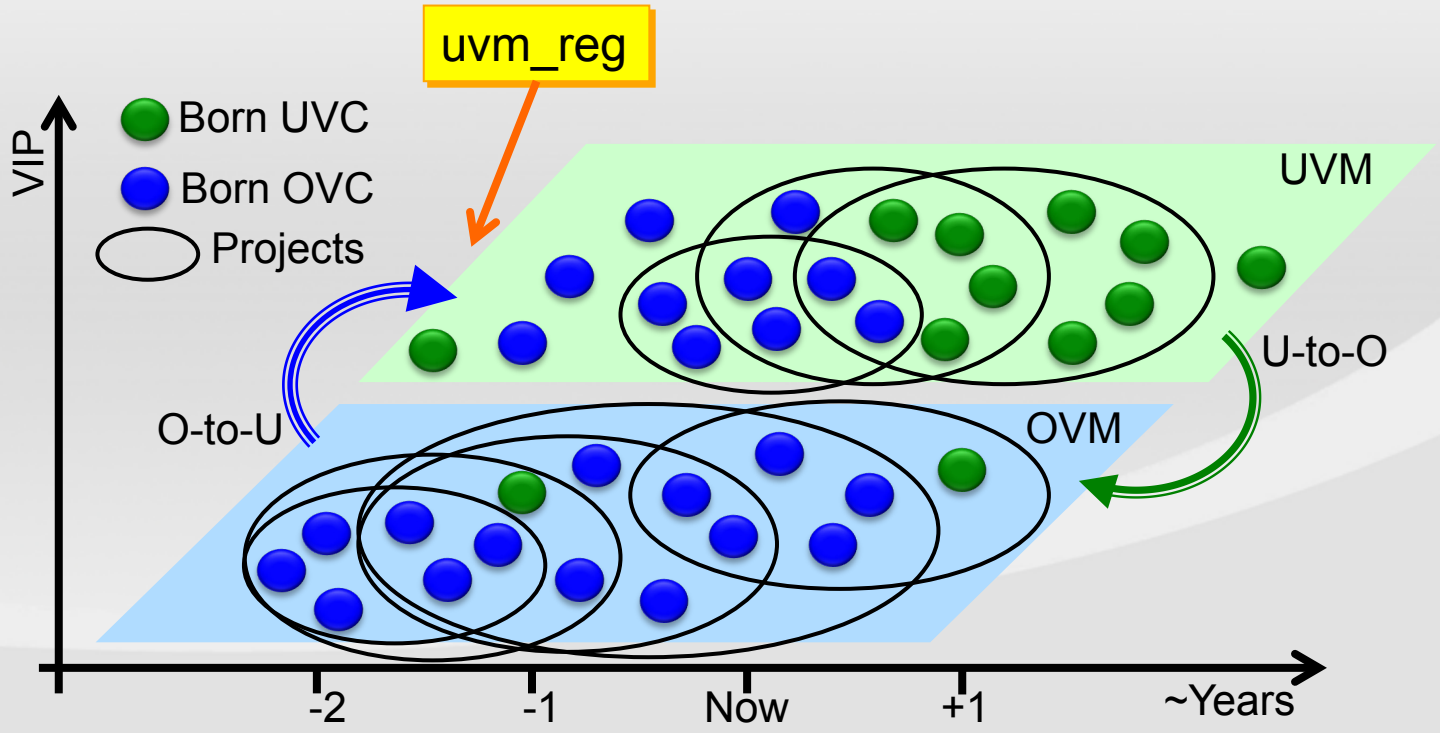
OVM to UVM Migration
or There and Back Again, a Consultant's Tale

Mark Litterick

Verification Consultant

Verilab GmbH, Munich, Germany

Transition from OVM to UVM

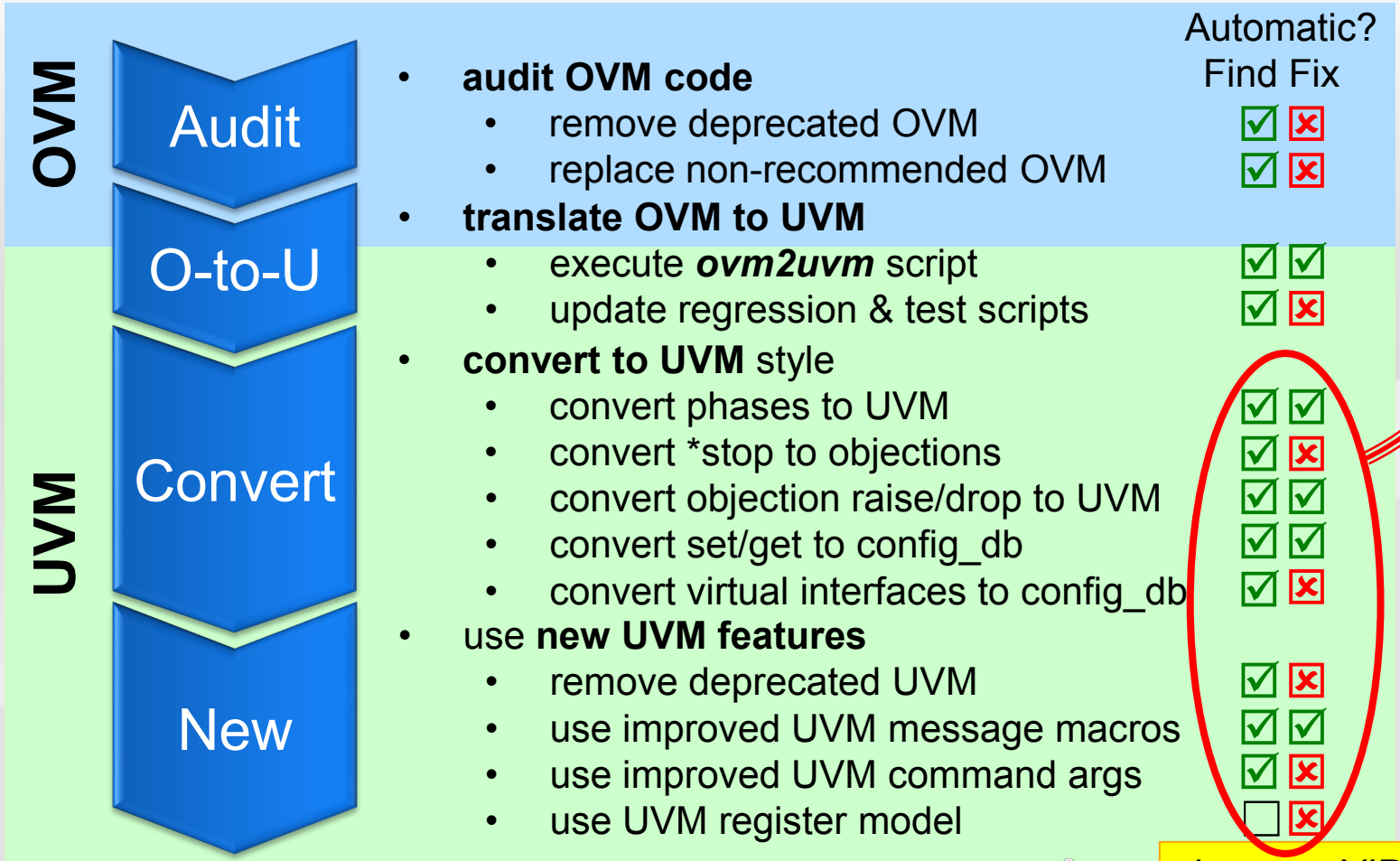


support **ongoing OVM** | provide **OVCs to UVM** | enable **new UVM** projects

Two Stage Evaluation

- **Starting point**
  - clean **OVM-2.1.2** OVCs
  - **no** AVM or URM **legacy**
  - different protocols but **common style** & development team
  - => scripts are more effective, but less general purpose!
- **First attempt – early UVM translation**
  - similar to Mentor Graphics' Verification Academy flow
  - goal : check for gotcha's in code, proof of concept, project running in UVM
- **Second attempt – late UVM translation**
  - optimized flow to do most effort in live OVM project
  - automatic and repeatable translation for delivery to UVM

Early UVM Translation



flow **works!**

too many **changes** after translation

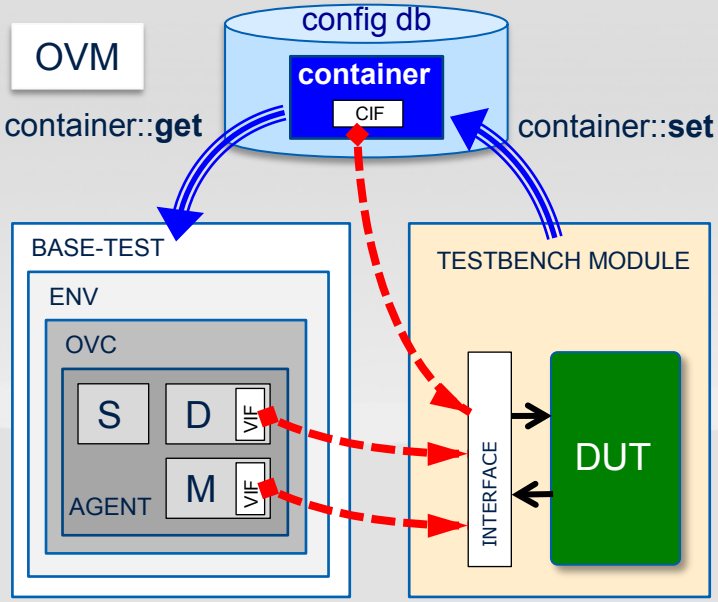
since our VIP from **live OVM** projects

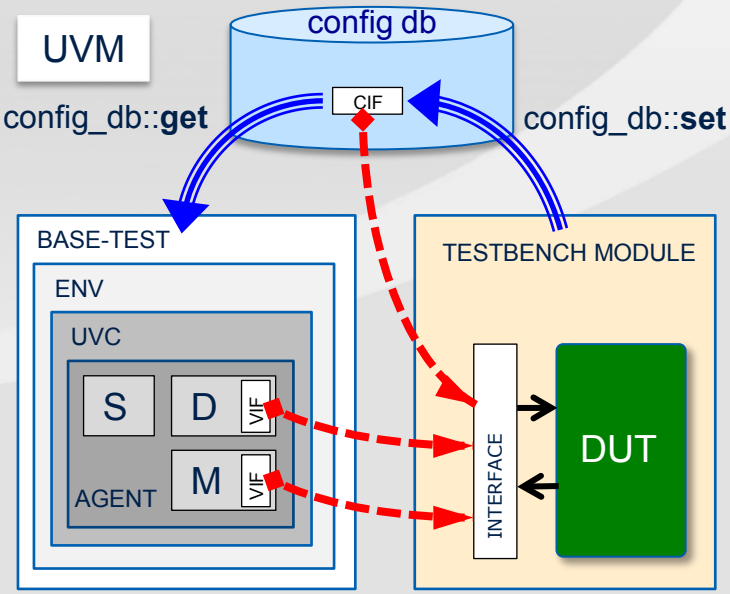
Push Back to OVM

- **Features already in OVM-2.1.2 source:**
  - only **objection handling** used for end-of-test
  - improved **message macros** used instead of methods
  - using **back-ported *uvm_reg*** in OVM environments
- **Key improvements that can be done in OVM:**
  - virtual **interface configuration**
  - **deprecate *sequence utils*** and OVM sequence library

Example: Interface Configuration

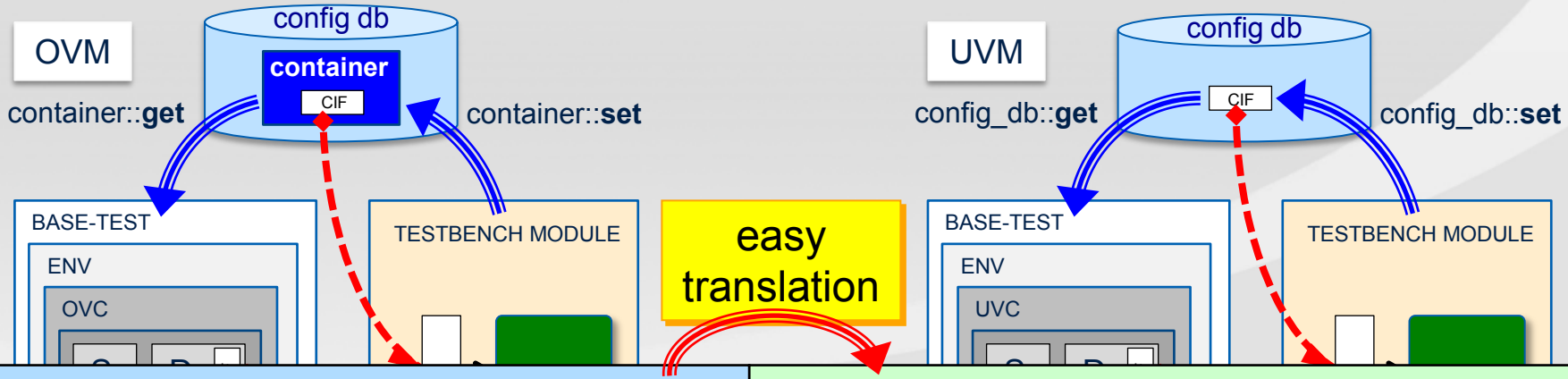
- **Improve virtual interface configuration**
  - better interface container – specifically to help OVM to UVM translation
  - container *set* and *get* methods similar to *uvm_config_db*





Example: Interface Configuration

- **Improve virtual interface configuration**
  - better interface container – specifically to help OVM to UVM translation
  - container *set* and *get* methods similar to *uvm_config_db*




```
// example set in OVM testbench module
my_container#(virtual my_if)::set("*", "cif", mif);

// example get in OVM agent or test class
if (!my_container#(virtual my_if)::get(this, "cif", vif))
  `ovm_fatal("NOVIF","...")
```

```
// example set in UVM testbench module
uvm_config_db#(virtual my_if)::set(null, "*", "cif", mif);

// example get in UVM agent or test class
if (!uvm_config_db#(virtual my_if)::get(this, "", "cif", vif))
  `uvm_fatal("NOVIF","...")
```

Example: Deprecated UVM

- **Sequencer & sequence *utils* deprecated in UVM**
  - OVM sequence library **not required** in UVM or OVM
  - **automatic script** & **manual repair** (for reactive slaves)

```
class my_seq extends ovm_sequence #(my_seq_item);
  `ovm_sequence_utils(my_seq, my_sequencer)
```

```
class my_sequencer extends ovm_sequencer #(my_seq_item);
  `ovm_sequencer_utils(my_sequencer)
  `ovm_update_sequence_lib_and_item(my_seq_item)
```

```
class my_env extends ovm_env;
  set_config_int("*.my_sequencer", "count", 0);
  set_config_string("*.my_sequencer","default_sequence","my_seq");
```

```
`ovm_do_on(my_seq, my_env.my_sequencer)
my_seq.start(my_env.my_sequencer);
```

Example: Deprecated UVM

- **Sequencer & sequence *utils* deprecated in UVM**
  - OVM sequence library **not required** in UVM or OVM
  - **automatic script** & **manual repair** (for reactive slaves)

can be done in **OVM or UVM**

do **once** in **OVM**

```
class my_seq extends ovm_sequence #(m
  `ovm_sequence_utils(my_seq, my_sequ
```

```
class my_seq extends ovm_sequence #(my_seq_item);
  `ovm_object_utils(my_seq)
  `ovm_declare_p_sequencer(my_sequencer)
```

```
class my_sequencer extends ovm_sequen
  `ovm_sequencer_utils(my_sequencer)
  `ovm_update_sequence_lib_and_item(my_seq_item)
```

```
class my_sequencer extends ovm_sequencer #(my_seq_item);
  `ovm_component_utils(my_sequencer)
```

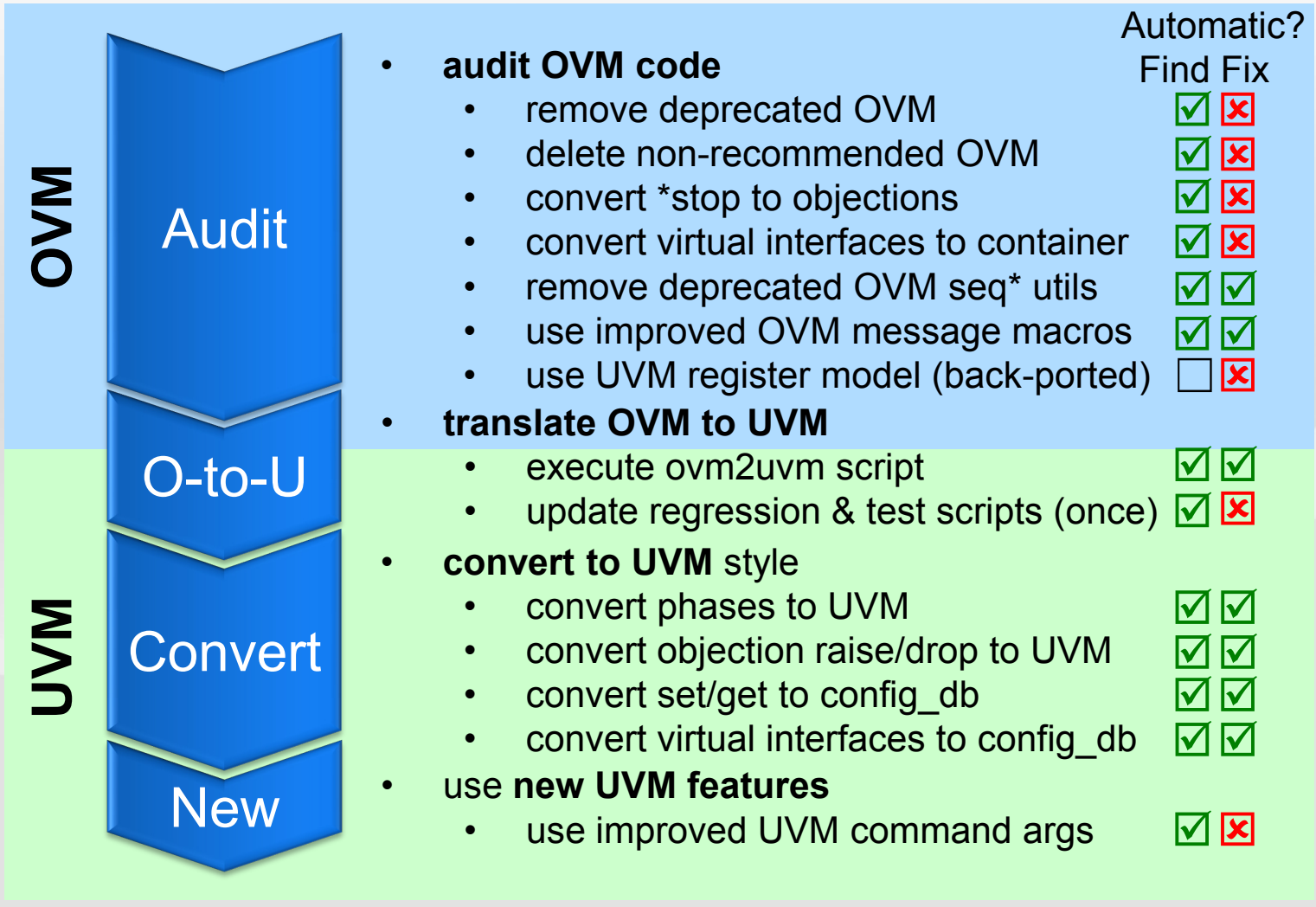
```
class my_env extends ovm_env;
  set_config_int("*.my_sequencer", "count", 0);
  set_config_string("*.my_sequencer","default_sequence","my_seq");
```

```
class my_env extends ovm_env;
```

```
`ovm_do_on(my_seq, my_env.my_sequer
my_seq.start(my_env.my_sequencer);
```

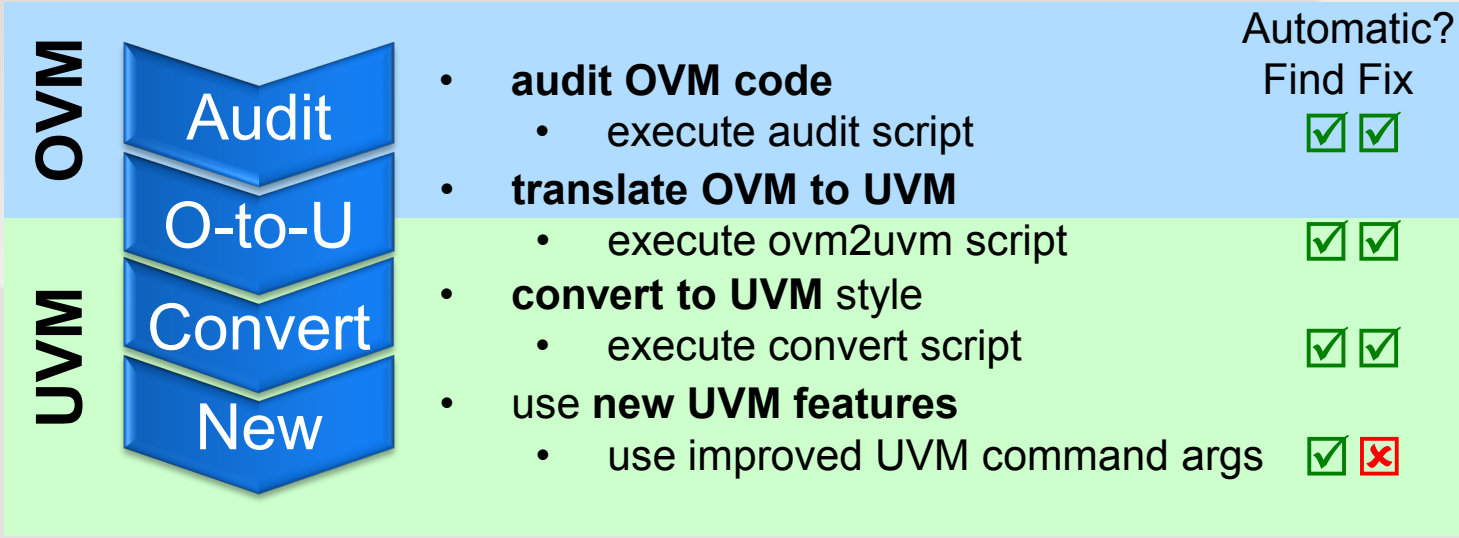
```
`ovm_do_on(my_seq, my_env.my_sequencer)
my_seq.start(my_env.my_sequencer);
```

Late UVM Translation



Final Translation Process

- **Prepare source OVC for translation – once**
- **Continue development of OVC in live OVM project**
- **Release OVC versions to UVM when appropriate**
- **Automatic translate to UVM as part of VIP release**



UVM to OVM Back-Porting

- **Slim OVM to UVM conversion supports reverse translation**
- **Valid when UVM transition period expected to endure**
- **Translate new UVC to OVC for ongoing OVM projects**
- **UVM limitations (hard to back-port)**
  - avoid **run-time phases**
  - avoid **TLM-2.0**
- **Other considerations (easier to back-port)**
  - modified objection handling
  - updated phase methods
  - config_db changes
  - command line processor

still **no industry consensus**

use **sequence-based phasing**

localize TLM2 if really required

OK if no run-time phases

normally OK

goal is **not to cripple UVM**
but enable **reuse in OVM**

Conclusion

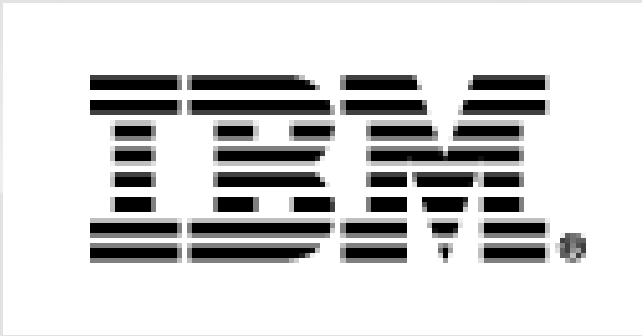
- **Goal is move to UVM**
  - **transition period** could endure for some time
  - considerable **OVM legacy** and many ongoing projects
  - **new UVM** projects **need OVC** libraries
  - **ongoing OVM** projects may need **new UVCs**
- **Presented an overview of migration process**
  - **prepare OVM** for easier translation
  - **slim automatic translation** process
  - translation process is **reversible**
- **Developed on family of OVCs, several projects**
  - applied to multiple projects @ different clients

mark.litterick@verilab.com

IBM Recommendations for OVM → UVM Migration

Wes Queen

Verification Manager, IBM



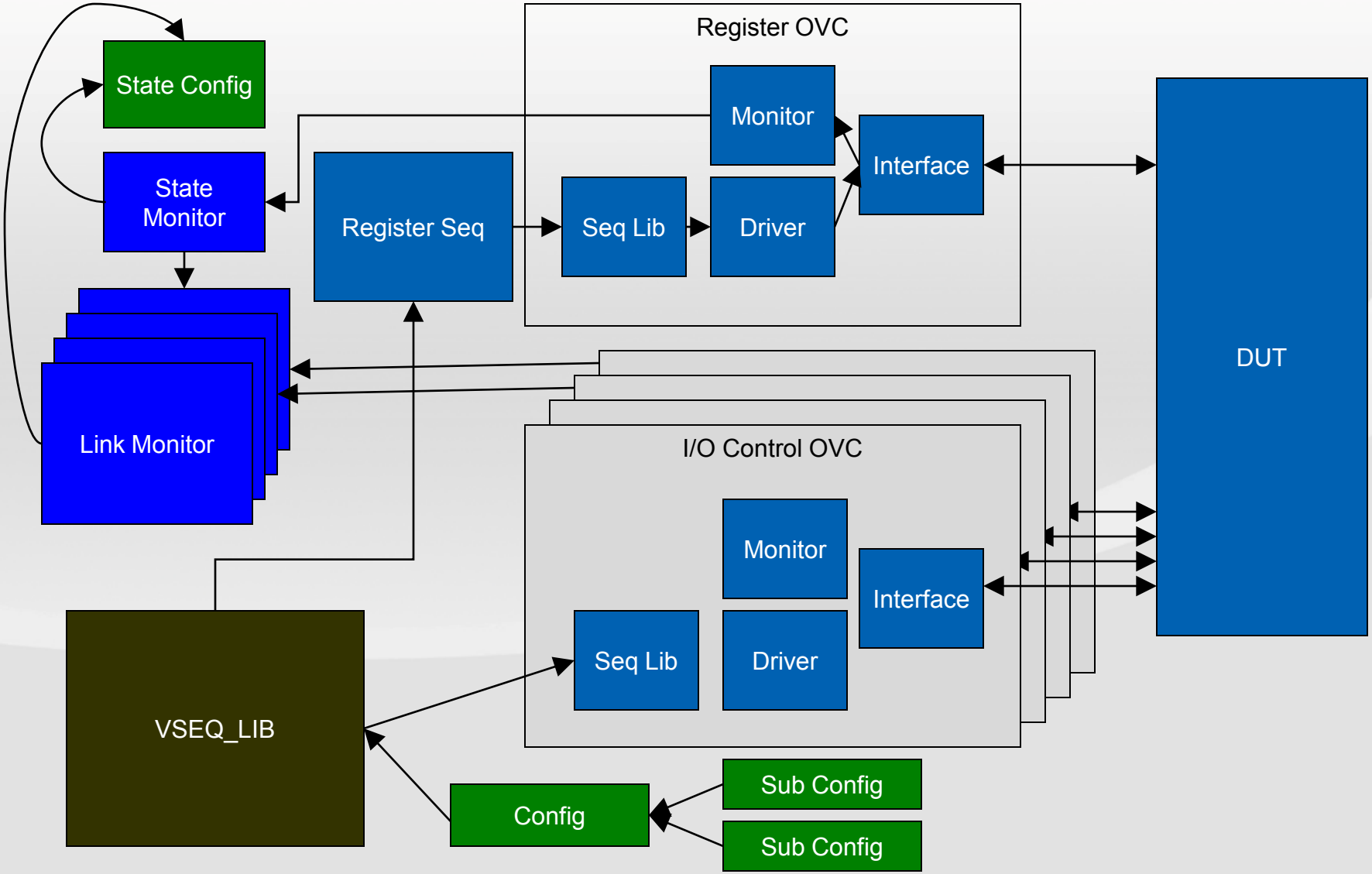
Migrating from OVM to UVM

- Motivation: UVM API beginning to diverge from OVM as new features are added to UVM
- Challenge: large code base in multiple projects
- General approach: Convert code base using scripts

OVM Development Started in 2009

- Open source ran on multiple simulators
- Methodology met verification team requirements for reuse
- Initial development followed user guide
- OVM_RGM register package adopted
- OVM use rapidly spread to multiple groups worldwide

Block Diagram HSS OVM Environment



Prepare Register Package

- **Install updated parser from Cadence**
  - Allows for OVM_RGM, UVM_RGM, and UVM_REG generation
  - OVM_RGM and UVM_RGM usage is identical
  - UVM_REG targets the Accellera package
- **Generate OVM and UVM register models**
  - Internal script used to generate file names and headers for new parser to match previous parser version
- **Install OVM_RGM 2.5 to align with new UVM parser**
  - Rerun OVM environment to be sure results match before proceeding

Run UVM Conversion Script

- **Download conversion guide from UVMWorld**
  - http://www.uvmworld.org/contributions-details.php?id=107&keywords=Appnote:_Migrating_from_OVM_to_UVM-1.0
  - Posted by John Rose on May 9, 2011 if you are navigating to find it
- **Install UVM conversion script**
  - Available within latest UVM kits on Accellera.org or within Cadence installation'
- **Move any directories out of code tree that should not be converted**
  - OVM_RGM directory and/or legacy code
- **Run conversion script**

Remove Deprecated Code and Compile DPI

- **Change any deprecated code which wouldn't compile (OVM 1.0, typically)**
  - Add_seq_cons_if – artifact code from OVM 1.0 that needs to be removed
  - Review conversion guide for other deprecated code

- **Compile uvm_dpi.cc (libdpi.so) in 32 bit or 64 bit**
  - New requirement for UVM

Golden Test and Further Updates

- **Run simulations to test conversion**
  - Include +UVM_USE_OVM_RUN_SEMANTIC in simulation command
- **On-going clean-up**
  - Remove other deprecated OVM calls (mostly super.build or straight build calls)
- **Adopt new UVM features**
  - Phases, sequences, UVM_REG, etc.

Results

- **Conversion process has been used successfully in multiple groups**
  - Current 4 projects have converted over the last year.
- **Effort was relatively low**
  - Lowest risk is to do the conversion between projects
  - Effort to convert took one day by single resource. 100K lines of code on single project.
- **Motivation to be on UVM is real**
  - New UVM features are valuable – UVM_REG, phasing, sequences, etc.
  - New UVM features can impact backward compatibility

FPGA chip verification using UVM

Ravi Ram

Principal Verification Engineer

Altera Corp

Charles Zhang

Verification Architect

Paradigm Works

Outline

- Overview
  - Verilog based verification environment
  - Why UVM?
  - New UVM based verification environment
  - FPGA chip verification flow
- Some of the challenges and solutions
  - Generic programmable logic
  - Legacy code integration.
  - Programmable core & IO connection
  - VIP integration(external and internal)

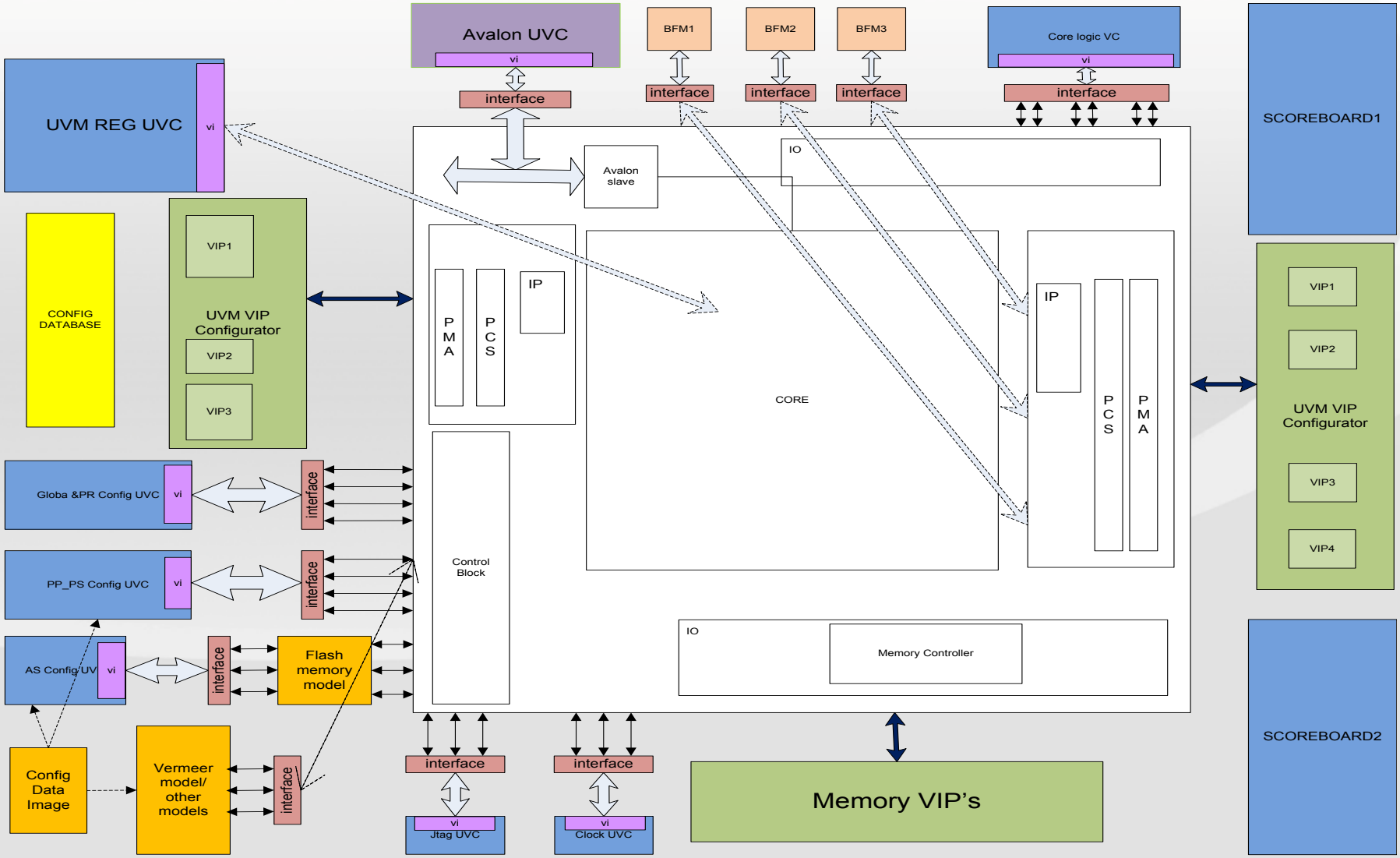
Verilog based Verification Env

- **Traditional Verilog based verification environment**
- **Multiple test benches for multiple modes of operation**
  - PP, PS, SPI, USERMODE, etc.
- **Recompilation for each test**
- **No object oriented programming (reuse = copy and change)**
- **Maintainability and scalability are poor (large number of tests, etc.)**
- **Easier for designer to understand and modify**

Why UVM?

- Supported and released by Accellera
- Supported by all major EDA vendors
- Object orient programming
- Reusability (vertical and horizontal)
- Well defined base class library
- Industry standard makes integration of third party or home grown VIP easier
- Good online documentation + UVM forums etc
- Little bit harder for designer to understand

UVM based Verification Env Overview



UVM-based verification Env overview

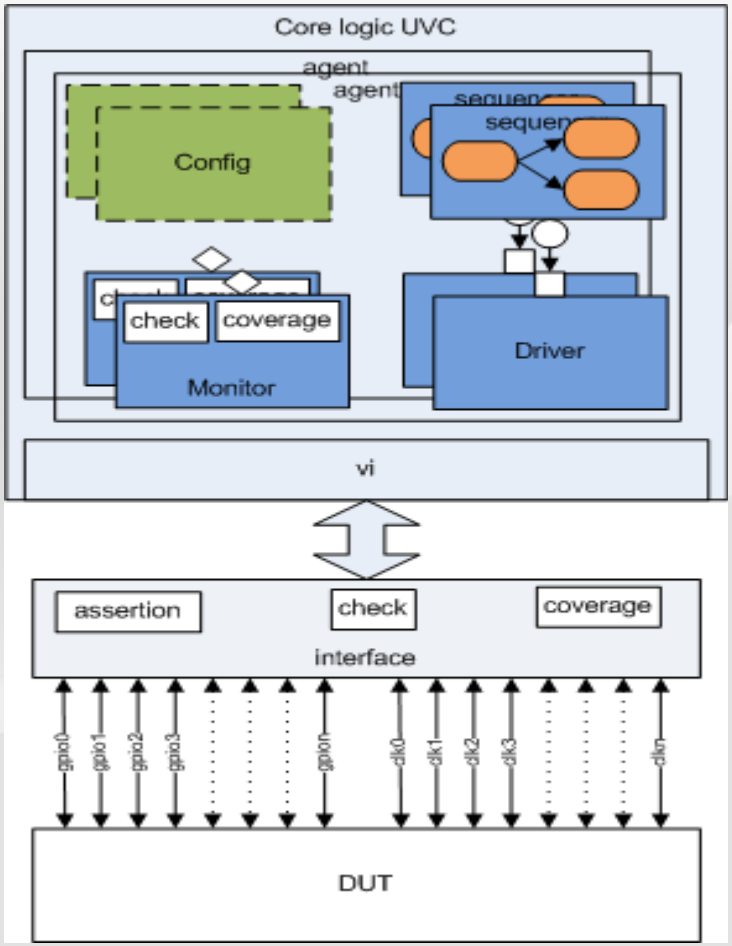
- Architected from scratch
- One environment supports multiple operating mode
  - PP, PS, SPI, USERMODE, etc.
- Significantly reduced number of tests by inheritance, configuration setting, etc
  - The current UVM based tests is about 1/3 of the tests of Module based ENV
- Simulation performance improved by compile once and run multiple tests
- Improved compile, run and regression flow
  - With UVM, cmd line processor is built-in and free

FPGA Verification Flow

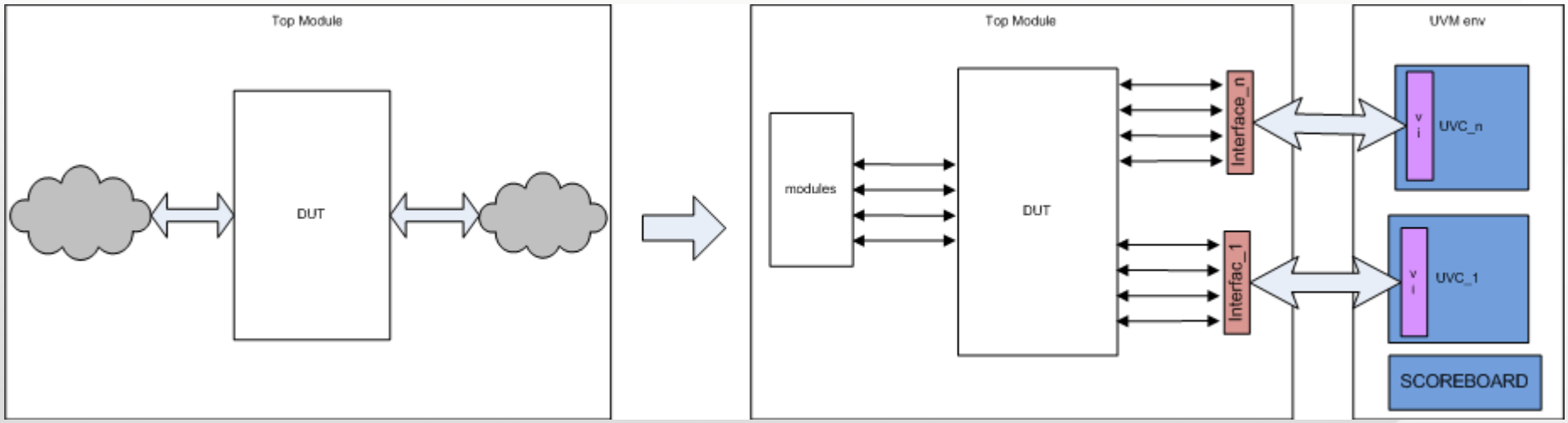
- Configuration (Programming the FPGA).
  - Support multiple programming interfaces
  - Data compression and encryption
  - Front door and back door loading configuration
  - Verification goal: make sure the programmed image matches the expected image
- User Mode (Running programmed user logic)
  - Tests include testing all core logic blocks and all the IO systems
  - Considerable effort is on creating configurable verification environment
  - Verification goal: verify all the core blocks and I/O systems to be functioning and connected properly

Generic programmable logic

- Programmable nature of FPGA calls for programmable verification environment
- Core logic interface UVC is a highly programmable verification component.
  - Allows user to decide on which pins to drive using UVM configuration
  - The monitor extended by user to implement any checking mechanism using UVM factory override.
  - Test based on sequences and transactions without worry about pin connection and toggling.
  - Compile once and run all tests.
- Used by the software group to verify real customer design.



Legacy code integration



- Still need Verilog based verification environment to coexist with UVM verification environment
- Interface file used as bridge between UVM verification environment and module based verification environment
- Interfaces bound to its physical interface signals
- Virtual interface in UVC set by getting the instance from resource database
- Assertions implemented in interface binds to module or physical interface signals

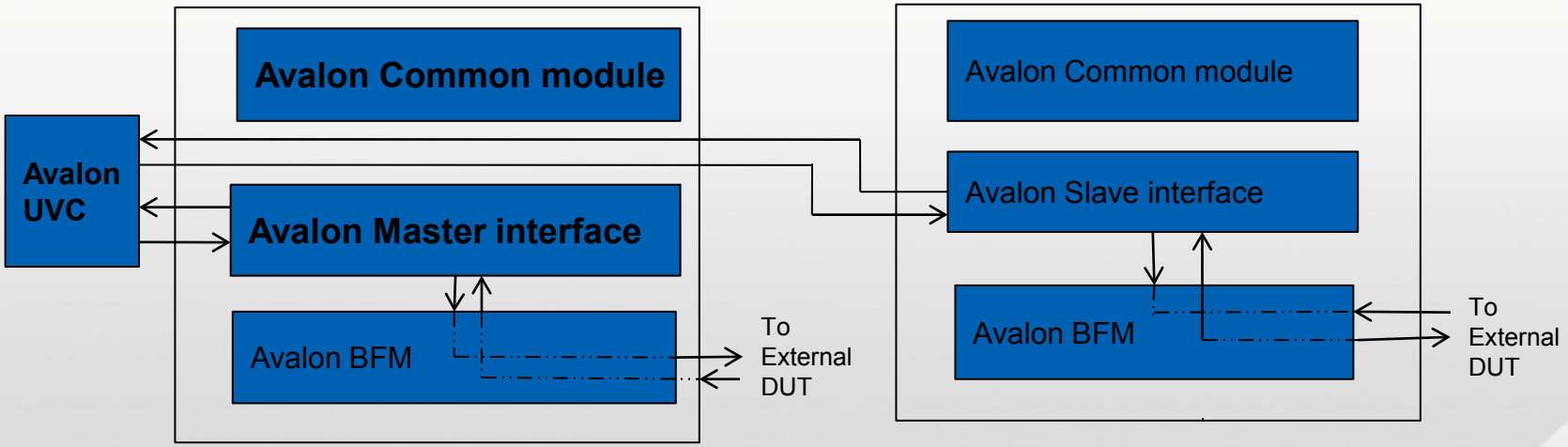
Programmable core & IO connection

- FPGA core is programmable
- All hard IP is configurable
- Lots of different interfaces and VIPs
- Register access from reg UVC to configure FPGA
  - Thousands of configurations in FPGA. UVM Reg model is already > 20G for handling 30 to 40% of the FPGA configurations. So this is not scalable and not practical to use
- Hundreds of configurable registers which UVM reg based testing cannot handle
  - Use home grown resource allocator plus configuration settings
- Register access from reg UVC to configure FPGA
- Seamless integration of resource allocator(internal tool) with internal developed tools for unidirectional and bidirectional connections

VIP integration

- **Lots of VIPs to address hard IP in FPGA(1G/10G…, PCIe plus other serial protocols, Altera Avalon VIP, different memory VIP for different memory protocols)**
- **Flexibility to configure and select VIPs in UVM test**
- **Use constraints to select the connections and VIPs**
- **Use on the fly point-to-point connections to connect VIP to the fabric**
  - Turn off unused VIPs
- **Same environment for integrating different vendor VIPs**
- **Environment setup for proliferation products for same FPGA family**
- **VIP interface easily portable to future FPGA families**

Avalon VIP Integration



- Integrate Avalon BFM in UVM environment
- Use of the existing bfm with a wrapper on top to make it a UVC
- VIP developed internally in Altera and is made available for use by all groups
- The configuration object generated for each instance of the VIP with a unique hdl Path which has a reference of the interface.
- The user provides the parameters for the VIP and the hdl Path in his test-bench hierarchy

Summary

- Altera's first verification project adopting UVM
- Addressed critical challenges
  - Programmable user logic and io
  - Explosive configuration spaces, etc.
- Adopted pragmatic view of the methodology
  - Re-architected the whole environment using UVM
  - Reused and integrated both internal and external VIPs
- UVM provides ideal way to create configurable, reusable verification components and environment

Q & A

Thank You!

Contributor: Manish Mahajan