A Reusable Verification Testbench Architecture Supporting C and UVM Tests

Richard Tseng
Qualcomm

Agenda

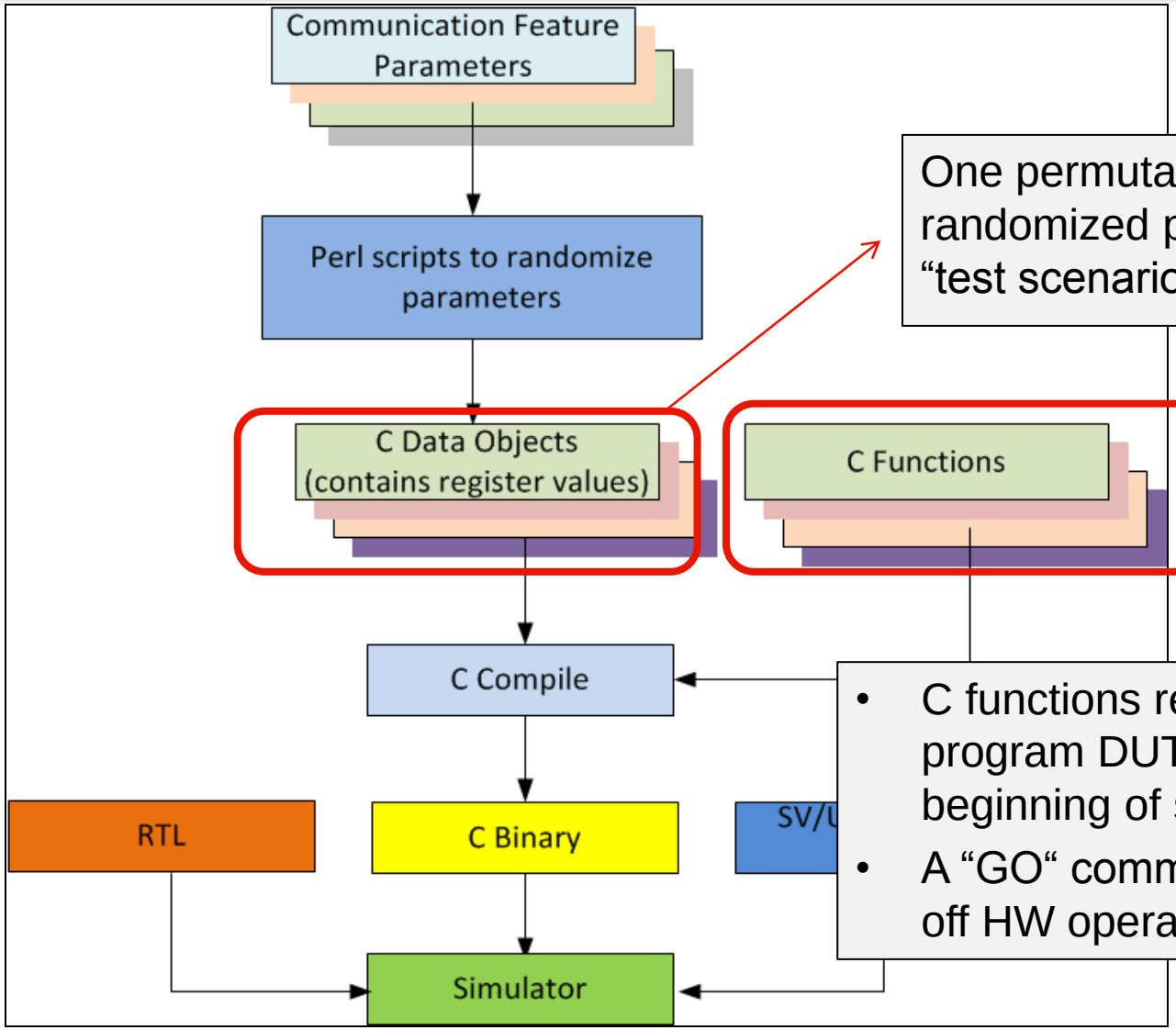
- Background & Requirements
- Testbench Architecture Overview
- Challenges & Solutions
- Summary
- Q&A

Background & Requirements

Background

- C was primarily used for modem mission mode testing
  - Thousand lines of legacy tests written in C
  - Hundred of scripts have been created to auto generate C tests
  - C test code and scripts have evolved over many product generations
  - They have been proven as **“golden”** regression suite
  - Randomizations were performed with Perl scripts
    - Randomized results are stored in multiple C data objects
    - C functions are called to access C data objects
- SV constraint solver is still used for bus interface testing & protocol compliance checking
- System engineering team (6 senior engineers) provides reference models and “test scenarios”

Test Scenario Generation



One permutation & combination of randomized parameters is called a "test scenario"

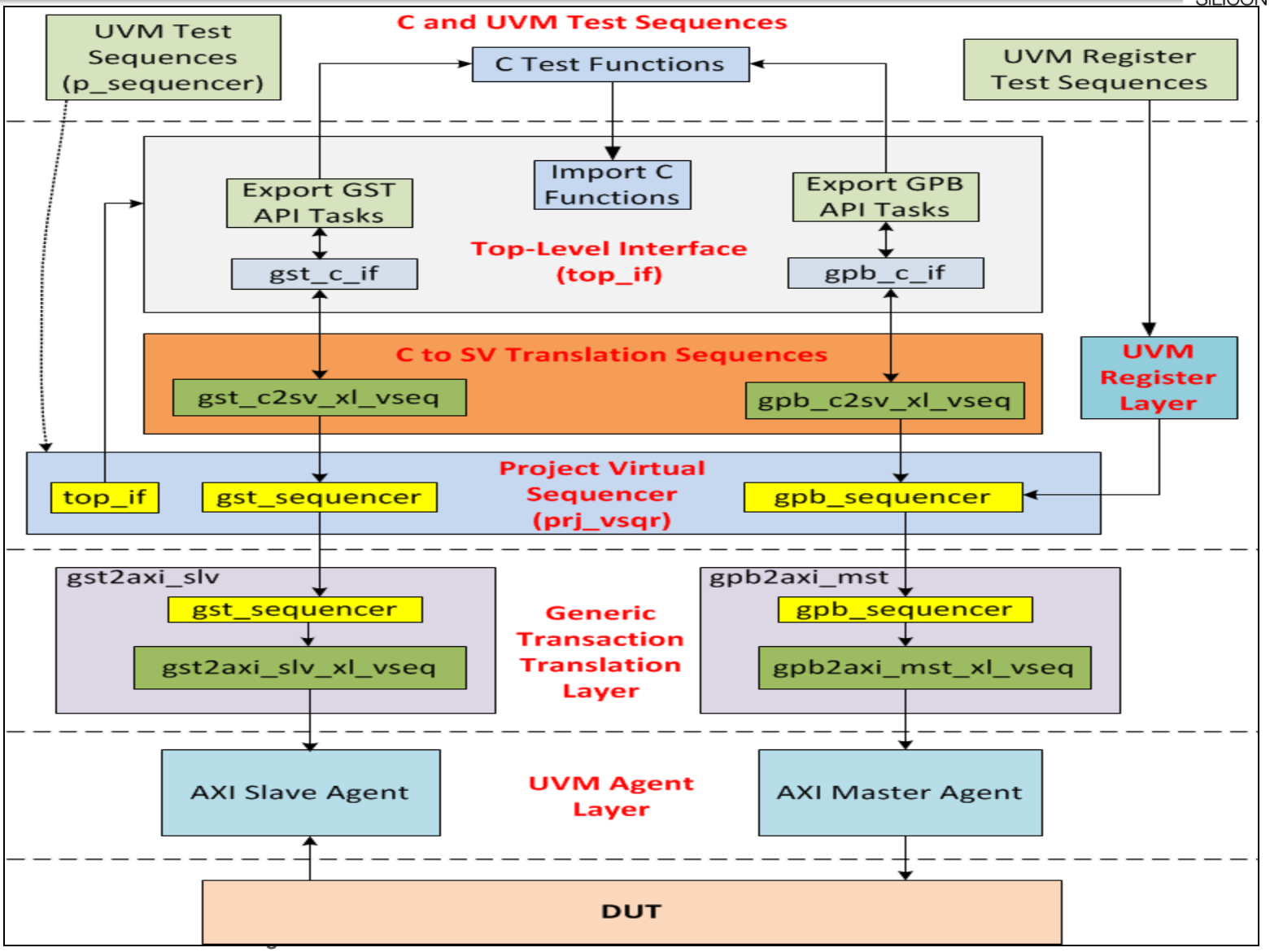
- C functions read data objects and program DUT register at beginning of simulation
- A "GO" command is issued to kick off HW operation

Verification Requirements

- Use UVM to build testbench
  - It's the latest and greatest technology
  - Strong EDA vendor support
  - Testbench components and test sequences can be reused
- Use API tasks to create C and UVM tests
- Reuse C and UVM tests in future design generations and various testbench platforms
- Run C and UVM tests simultaneously
- Migrate all legacy C tests to UVM sequences
- Integrate UVM Register Layer

Testbench Architecture Overview

Testbench Architecture Overview



Challenges & Solutions

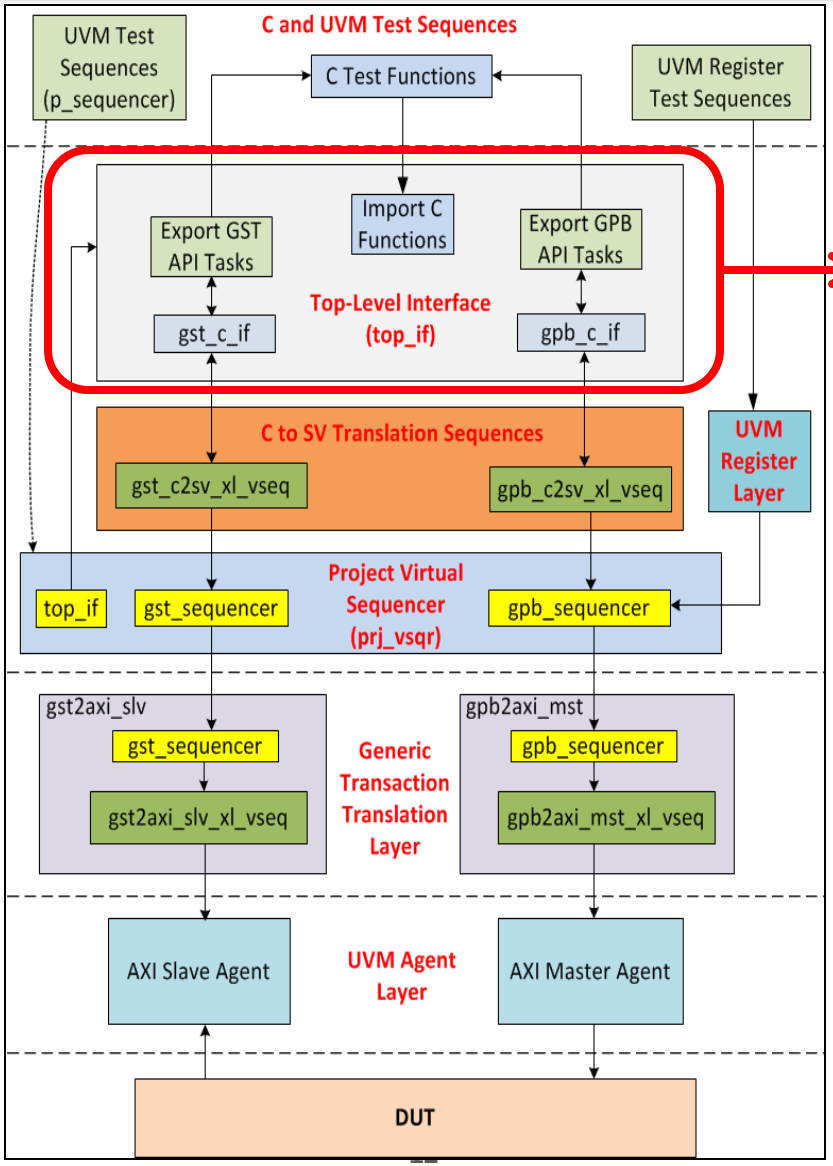
Challenges & Solutions

1. Building C-SV communication interface
2. Driving test flow with C and/or UVM sequences
3. Creating UVM sequences with API Tasks
4. Reusing high-level C and UVM tests and testbench components

Challenges & Solutions

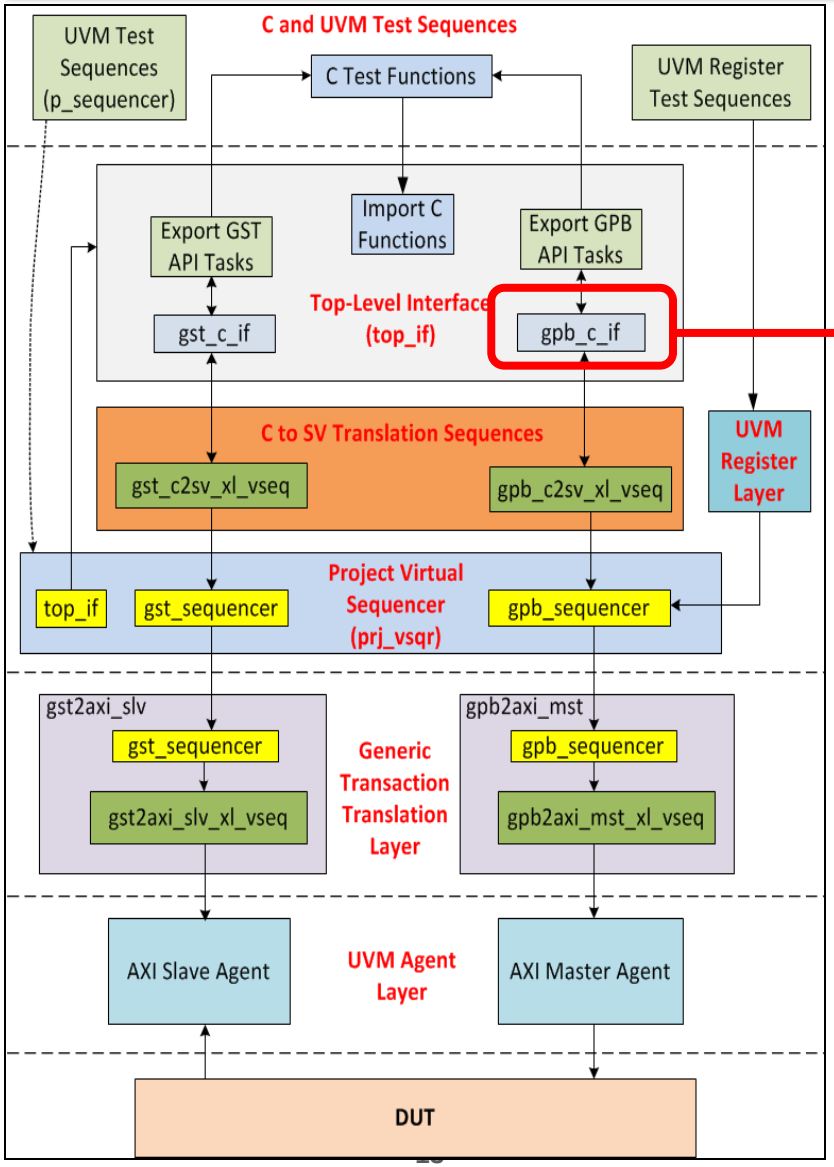
1. **Building C-SV communication interface**
2. Driving test flow with C and/or UVM sequences
3. Creating UVM sequences with API Tasks
4. Reusing high-level C and UVM tests and testbench components

Top-level interface - top_if



- Define a set of SV API tasks for each generic interface
- Export SV API tasks to C domain
- Import C test functions
- Define sub-interface (interface of Interface)
  - Each sub-interface communicates with a C-SV translation sequence and a set of API tasks
  - Add semaphore and synchronization events to handle the race conditions
- Instantiate multiple sub-interfaces

Sub Interface - gpb_c_if.sv




```
interface gpb_c_if();
  import gpb_pkg::*;
  gpb_op_type cmd;
  bit[31:0] addr;
  bit[31:0] data;
  int unsigned nop_cycle;

  // synchronization events and
  // semaphore
  event       cmd_rdy;
  event       cmd_done;
  semaphore   tr_sm;

endinterface: gpb_c_if
```

Top-level interface (top_if.sv)


```
interface top_if(…);
  // 1 - Instantiate sub interface (gpb_c_if)
  gpb_c_if gpb_c_if0();
```

```
  // 2 - Set sub interface and allocate semaphore
  initial begin
    uvm_config_db#(virtual gpb_c_if)::set(uvm_root::get(),"*",
                   "gpb_c_if0", gpb_c_if0);
   gpb_c_if0.tr_sm = new(1); // allocate 1 semaphore key
 end
```

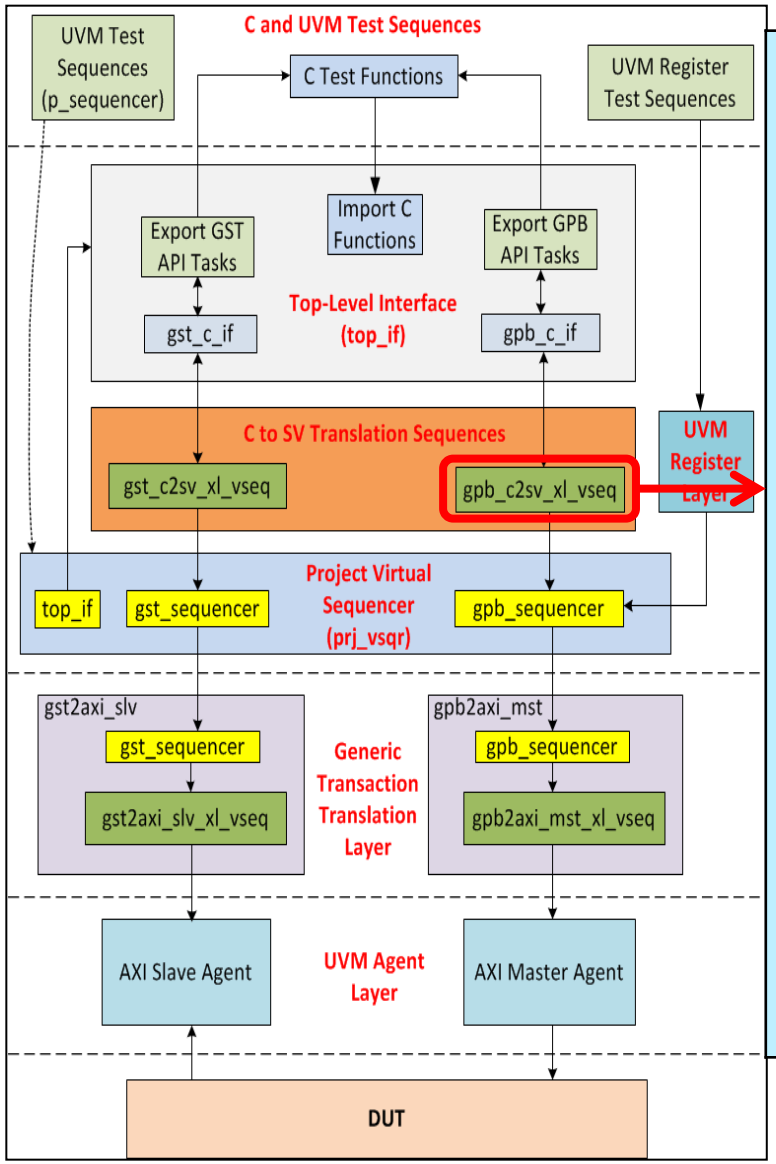
```
  // 3 - Define API task
  task automatic gpb_write_dpi(input int unsigned addr,
                               input int unsigned data);
    …
  endtask
```

```
  // 4 - Export API task to C
  export "DPI-C" gpb_write = task gpb_write_dpi;
endinterface
```

Define API task


```
task automatic gpb_write_dpi(input int unsigned addr,
                             input int unsigned data);

  gpb_c_if0.tr_sm.get(1); // blocks until get one key

  // perform write operation
  gpb_c_if0.cmd     = gpb_pkg::WRITE;
  gpb_c_if0.addr    = addr;
  gpb_c_if0.wr_data = data;

  ->gpb_c_if0.cmd_rdy; // start the write operation
  @gpb_c_if0.cmd_done; // wait for translation seq trigger

  gpb_c_if0.tr_sm.put(1); // put a key back

endtask: gpb_write_dpi
```

C-SV Translation Sequence




```
`define intf p_sequencer.gpb_c_if0

class gpb_c2sv_xl_vseq extends
                              gpb_base_vseq;
  task body();
    forever begin
      @(`intf.cmd_rdy);
      case (`intf.cmd)
        WRITE: begin
          gpb_write(`intf.addr, `intf.data);
        end
        READ: begin
          gpb_read(`intf.addr, `intf.data);
        end
        …
      endcase
      -> `intf.cmd_done;
    end
  endtask: body
endclass: gpb_c2sv_xl_vseq

`undef intf
```

Launch C-SV translation sequence

- Translation sequences are created in the base test, and forked-off at the beginning of a simulation

```
class base_test extends uvm_test;
  virtual task run_phase(uvm_phase phase);
    fork
      begin: start_gpb_c2sv_xl_vseq
        gpb_c2sv_xl_vseq vseq0;
        vseq0 = gpb_c2sv_xl_vseq::type_id::create(…);
        vseq0.start(vsqr.gpb_sqr);
      end: start_gpb_c2sv_xl_vseq

      begin: start_gst_c2sv_xl_vseq
        gst_c2sv_xl_vseq vseq0;
        vseq0 = gst_c2sv_xl_vseq::type_id::create(…);
        vseq0.start(vsqr.gst_sqr);
      end: start_gst_c2sv_xl_vseq
    join_none
  endtask: run_phase
endclass: base_test
```

Challenges & Solutions

1. Building C-SV communication interface
2. **Driving test flow with C and/or UVM sequences**
3. Creating UVM sequences with API Tasks
4. Reusing high-level C and UVM tests and testbench components

Drive test flow with C or UVM sequences

- All tests are launched from UVM - **sequence.start(sequencer)**
- C test can be launched as below:

```
interface top_if();
  export "DPI-C" gpb_write = task gpb_write_dpi;
  export "DPI-C" gpb_read  = task gpb_read_dpi;
  export "DPI-C" gpb_nop   = task gpb_nop_dpi;
  import "DPI-C" c_test    = task c_test_dpi;
endinterface;

class my_sv_seq extends prj_base_seq;
  virtual task body() begin
    // p_sequencer has the handle of top_if
    // top_if has the c_test imported
    p_sequencer.top_if0.c_test_dpi();
  endtask: body
endclass
```

```
void c_test() {
  gpb_write(addr1, data1);
  gpb_read(addr2, &data2);
  gpb_nop(10);
  …
}
```

Launch multiple C/UVM mixed tests


```
interface top_if(…); // import c_test1
  import "DPI-C" context c_test1 = task c_test1_dpi();
  …
endinterface

class multi_procs_vseq extends prj_base_sequence;
  task body();
    fork
      begin: proc_1 // launch C function
        p_sequencer.top_if0.c_test1_dpi();
      end
      begin: proc_2 // launch SV sequence
        my_sv_seq.start(p_sequencer);
      end
    join
  endtask: body
endclass: multi_procs_vseq
```

Challenges & Solutions

1. Building C-SV communication interface
2. Driving test flow with C and/or UVM sequences
3. **Creating UVM sequences with API Tasks**
4. Reusing high-level C and UVM tests and testbench components

Creating UVM sequences with API tasks

- C tests are usually programmed with API tasks
- Typically in an UVM testbench, UVM macros are used to create a test sequence
  - i.e. `uvm_do(),`uvm_do_with(), `uvm_do_on_with()
- UVM “do” macros don’t match the legacy C tests, using API tasks is more consistent
- To reuse C tests in an UVM testbench, C API tasks need to be mapped to UVM sequences
  - Constraints can be specified in API task arguments
  - Default constraints can be specified within the API task

UVM “do” macro versus API tasks

- 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 transaction using API task:

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

Create Transaction with API tasks


```
task axi_master_base_seq::axi_master_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_master_write
```

Create a sequence item

Randomize with address, data, and default constraints

Send to sequencer

Test Sequence Created with API Tasks


```
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(addr1, 32'h1234_5678);
  axi_slave_bkdr_read(addr2, read_data);
endtask
```

Challenges & Solution

1. Building C-SV communication interface
2. Driving test flow with C and/or UVM sequences
3. Creating UVM sequences with API Tasks
4. **Reusing high-level C and UVM tests and testbench components**

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


```
// 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(addr1, 32'h1234_5678);
  gst_bkdr_read(addr2, read_data);
endtask
```

Maps to:

```
// 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(addr1, 32'h1234_5678);
  axi_slave_bkdr_read(addr2, 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

Synopsys Users Group
SILICON VALLEY 2013

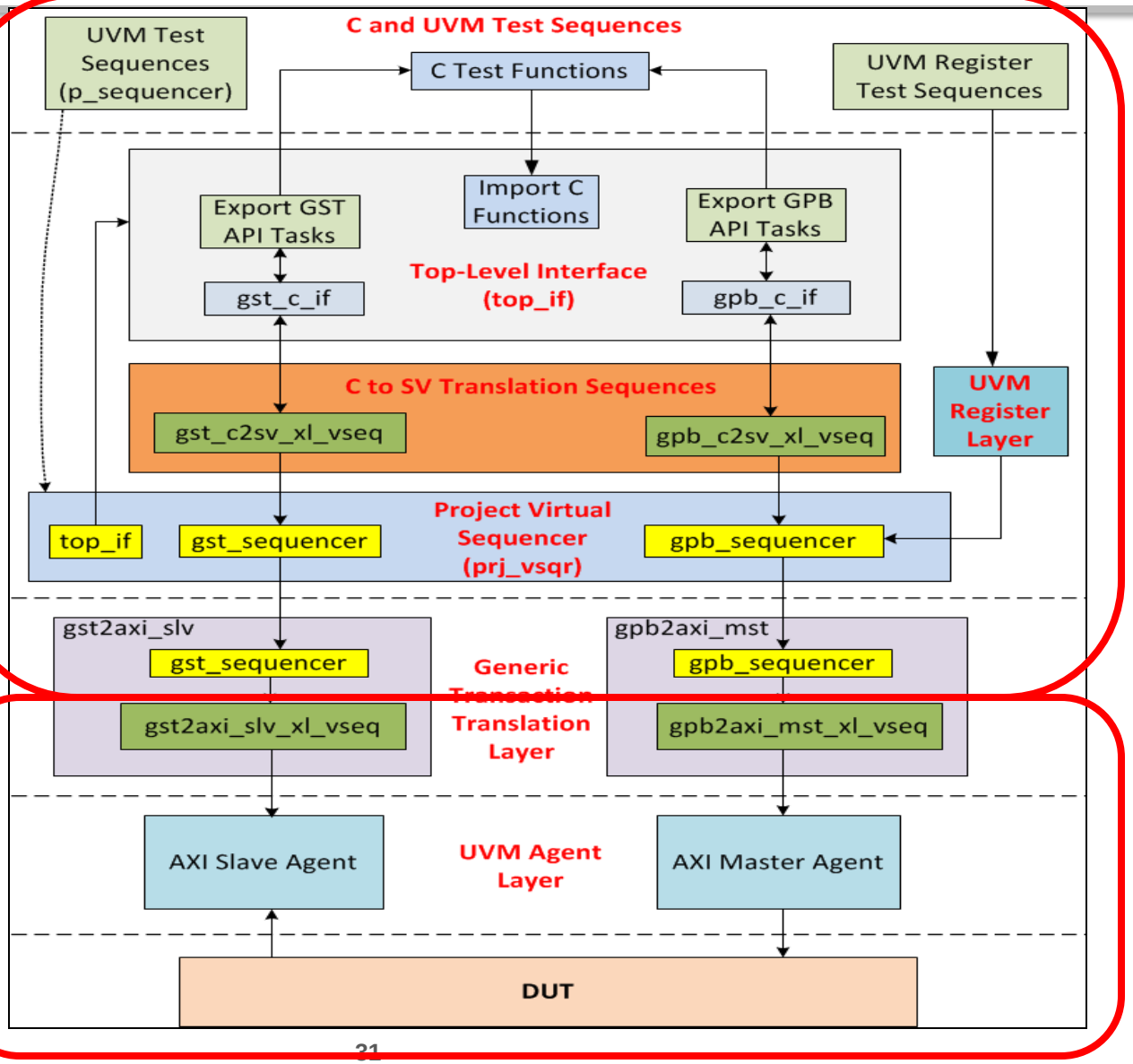
```
// In SV file, define uvm_rpt_info() function
function uvm_rpt_info(string id, string message,
                      int verbosity = UVM_MEDIUM);
  `uvm_info(id, message, verbosity)
endfunction: uvm_rpt_info
```

```
// 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 and test sequences



Reusable!

For different designs, replace:

- UVM Agent Layer
- Translation sequences in GTTL

Summary

Summary

- In our applications, test sequences and the VIPs were reused in multiple testbenches:
  - Modem core-level testbench
    - GPB => AXI master
    - GST => 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 – VCS is the best!!

- The testbench architecture extends reusability beyond the scope of the UVM technology, and across the C and SV language boundary

Q&A