UVM Made Language Agnostic: Introducing UVM For SystemC

Akhila M, Intel India Pvt. Ltd
Outline

Intro to UVM-SystemC

Pre-UVM-SystemC, Class Based Verification Environment

Migrating to UVM-SystemC
Intro to UVM-SystemC

• UVM-SystemC = UVM implemented in SystemC/C++
• No structured nor unified verification methodology available for ESL designs
• Porting UVM to SystemC/C++ enables
  – Re-use tests and test benches across verification (simulation) and validation (HW-prototyping) platforms.
  – Verification components can be re-used across block and system level.
• Target to make UVM truly universal, not tied to a particular language
UVM(UVM-SC) Layered Architecture

# Interfaces

# Drivers = # Interfaces
Pre-UVM-SystemC, Class Based Verification Environment

- Custom test functions & test engine
  - Test functions (test ports) created to mimic bus transaction drivers; one per transaction type
  - Individual functions for each test scenario
- Distinct methods for writes and reads
  - Number of methods depends on types of valid write/read as per the protocol being implemented viz. single, burst, posted or non-posted
- Self-checking and parametrized tests
- Additional task created for running multiple tests in parallel from different interfaces
  - Vector classes used to keep track of test functions being launched from each interface
class ahb_test_port: public test_port 
{
public:
  // Each test port has individual parameters
  typedef sc_uint<AhbAddrWidth> AhbAddr_t;
  typedef typename sc_suint<AhbDataWidth>::T AhbData_t;
  static const unsigned AHB_RESP_WIDTH = (AHB_FULL_PORT) 
    ? amba_sysc::AHB_FULL_RESP_WIDTH : 
    amba_sysc::AHB_LITE_RESP_WIDTH;
public:
  // AHB bus signals
  sc_signal<bool>                     hsel;
  sc_signal<AhbAddr_t>                haddr;
  sc_signal<sc_uint<amba_sysc::AHB_TRANS_WIDTH> > htrans;
  sc_signal<bool>                     hwrite;
  sc_signal<sc_uint<amba_sysc::AHB_SIZE_WIDTH> > hsize;
  sc_signal<sc_uint<amba_sysc::AHB_BURST_WIDTH> > hburst;
  sc_signal<sc_signal<AhbData_t>    hdata;
  sc_signal<bool>                     hready;
  sc_signal<AhbData_t>                hrdata;
  sc_signal<bool>                     hready_resp;
  sc_signal<sc_uint<AHB_RESP_WIDTH> > hresp;

  explicit ahb_test_port (const sc_module_name &name, FlexibleClk* f_clk, 
    const std::string& port_name, 
    int port_id) :
    test_port(name, port_name, port_id, 0, 0, f_clk)
  {}

virtual void reset()
{
  haddr.write(0);
  hsel.write(0);
  hwrite.write(0);
  htrans.write(0);
  hsize.write(0);
  hburst.write(0);
  hwdata.write(0);
}

virtual void b_write(ApiAddr_t addr, 
  ApiData_t data, 
  unsigned wid = 0, 
  unsigned wstrb = 0xFFFF, 
  unsigned burst_size = amba_axi::ASIZE_32, 
  unsigned checkResp = 0)
{
  // ADDRESS PHASE
  while (!hready_resp) wait();
  hsel.write(1);
  hready.write(1);
  haddr.write(addr);
  hwrite.write(1);
  htrans.write(amba_sysc::HTRANS_NONSEQ);
  hsize.write(burst_size);
  hburst.write(0);
  hwdata.write(0);
  wait();
  // DATA PHASE
  htrans.write(amba_sysc::HTRANS_IDLE);
  hwdata.write(data);
  wait();
  while (!hready_resp) wait();
}
class MultiPortTask {
    std::vector<Task*> tasks;

public:

    /**
     * Filled task object (however not yet launched) are added using this method
     * Each added task should have a port specified
     */
    MultiPortTask& add(Task& task) {
        assert(!task.was_run);
        // Check that there's no tasks with the same port in the queue
        for (int i = 0; i < tasks.size(); i++) {
            assert (task.port->getPortId() != tasks[i]->port->getPortId());
        }
        tasks.push_back(&task);
        return *this;
    }

    /**
     * Push each supplied test to a corresponding port
     * This method blocks until all test will finish.
     */
    void run() {
        // Wait for all involved ports to be idle, i.e. not having pending or
        // launched tests
        for (int i = 0; i < tasks.size(); i++) {
            while (tasks[i]->port->isWorking()) wait();
        }
        // Run tests
        for (int i = 0; i < tasks.size(); i++) {
            test_port::port_task tmp;
            tasks[i]->was_run = true;
            tmp.first = tasks[i]->test;
            tmp.second = tasks[i]->ctx;
            tasks[i]->port->task_fifo.write(tmp);
        }
        // Wait until all tests will finish
        for (int i = 0; i < tasks.size(); i++) {
            while (tasks[i]->port->isWorking()) wait();
        }
    }
Limitations and Potential Reasons for UVM-SystemC Adoption

• Limitations of Non-UVM-SC testbench
  – Higher learning curve for new user as TB has no standardized architecture
  – Minimal reuse of tests/components across projects
  – Configurability of testbench is limited
  – Inadequate constrained randomization
  – Narrow scope of IP to SoC reuse (SoC usually has UVM-SV based framework)

• Reasons for UVM-SC adoption:
  – Re-usability
  – Configurability
  – Constrained randomization
  – Standardization across languages
  – Easier adoption for UVM-SV users
Migrating to UVM-SystemC framework

- UVM-SystemC adheres to the UVM-SystemVerilog standard layered architecture
  - Migration of previous components to their respective layers required
Signal Layer

- Create interface and transaction classes as needed by the protocol
- Connect DUT to the interface
- Pass this interface to other components throughout the testbench hierarchy using configuration database(uvm_config_db)

```c
int sc_main(int, char*[]) {
    ahb_clk_reset_gen* clk_rst_gen
        = new ahb_clk_reset_gen("clk_rst_gen");
    ahb_if* dut_if_in = new ahb_if("dut_if_in");
    dut_if_in->hclk(clk_rst_gen->ahb_clk);
    dut_if_in->hresetn(clk_rst_gen->reset_val);
    dut ahb_dut("ahb_dut");
    ahb_dut.hclk(clk_rst_gen->ahb_clk);
    ahb_dut.hresetn(clk_rst_gen->reset_val);
    ahb_dut.haddr(dut_if_in->haddr);
    ...
    uvm::uvm_config_db<ahb_if*>::set
        (0, "*", "vif", dut_if_in);
    uvm::uvm_config_db<sc_event*>::set
        (0, "*", "reset_done", clk_rst_gen->reset_done);
    run_test("ahb_wr_rd_test");
    return 0;
}
```
Component Layer

- All components in this layer are mapped to a transaction type
  - The transaction class constitutes on the packet type which is being transmitted across components
- Driver-Sequencer to follow a standard handshaking interface as per UVM standard
- Driver is the key component where all protocol intelligence has to be implemented
- Monitor can implement protocol checks, data integrity checks etc.
  - Taps the DUT signal directly
class ahb_driver : public uvm::uvm_driver<ahb_transaction> {
    public:
        UVM_COMPONENT_PARAM_UTILS(ahb_driver);
        ahd_if* ahd_vif;
        sc_event* reset_event_driver;
        sc_semaphore ahd_pipeline_lock;

        ahd_driver( uvm::uvm_component_name name = "ahd_driver"):
            uvm::uvm_driver<ahb_transaction>( name ),ahd_pipeline_lock(1)
            { ... } 

    void build_phase(uvm::uvm_phase& phase) {
        UVM_INFO(this->get_name(), "build_phase Entered", UVM_LOW);
        uvm_driver<ahb_transaction>::build_phase(phase);
        reset_event_driver = new sc_event("reset_event_driver");

        if(!uvm_config_db<ahd_if*>::get(this, "/", "vif", ahd_vif)) {
            UVM_FATAL(this->get_name(),"AHB Virtual Interface missing");
        } 

        if(!uvm_config_db<sc_event*>::get(this, "/", "reset_done", reset_event_driver))
            { UVM_FATAL(this->get_name(), "Reset event missing");
        } 
    }
void run_phase(uvm::uvm_phase& phase) {
    UVM_INFO(this->get_name(), "run_phase Entered", UVM_LOW);
    if(ahb_vif->hresetn == 0) wait(*reset_event_driver);
    while(true) {
        SC_FORK
        sc_spawn(sc_bind(&ahb_driver::send_transaction, this), "drive1"),
        sc_spawn(sc_bind(&ahb_driver::send_transaction, this), "drive2")
        SC_JOIN
    }
    UVM_INFO(this->get_name(), "run_phase finished", UVM_LOW);
}
void send_transaction() {
    ahb_transaction req, rsp;
    ahb_pipeline_lock.wait();
    UVM_INFO(this->get_name(), "send_transaction: next item", UVM_LOW);
    this->seq_item_port->get_next_item(req);
    ahb_vif->htrans = req.htrans;
    ahb_vif->haddr = req.haddr;
    ahb_vif->hsize = req.hsize;
    ...
    wait(AHB_CLK);
    while (ahb_vif->hready != 1) wait(AHB_CLK);
    rsp.set_id_info(req);
    this->seq_item_port->item_done();
    this->seq_item_port->put_response(rsp);
    ahb_pipeline_lock.post();
}
class ahb_monitor : public uvm::uvm_monitor {
public:
  uvm::uvm_analysis_port<ahb_transaction> item_collected_port;
  ahb_if* vif;

  ahb_monitor(uvm::uvm_component_name name = "ahb_monitor"):
    uvm_monitor(name), item_collected_port("item_collected_port"), vif(0) {
    ...
  }

  UVM_COMPONENT_UTILS(ahb_monitor);

  void build_phase(uvm::uvm_phase& phase) {
    uvm::uvm_monitor::build_phase(phase);
    if (!uvm::uvm_config_db<ahb_if*>::get(this, ",", "vif", vif)) {
      UVM_FATAL(name(), "Virtual interface not defined!");
    }
  }

  void run_phase(uvm::uvm_phase& phase) {
    ahb_transaction pkt;
    while (true) { // monitor forever
      std::ostringstream str;
      wait(vif->hresetn.posedge_event());
      if(vif->hclk == 0) sc_core::wait(vif->hclk.posedge_event());
      pkt.htrans = vif->htrans;
      ...
      item_collected_port.write(pkt);
    }
  }
}
#include "ahb_wr_rd_sequence.h"

class ahb_wr_rd_test : public uvm::uvm_test
{
public:
    ahb_wr_rd_sequence* ahb_wr_rd_seq;
    ahb_basic_env* top_env;

    UVM_COMPONENT_UTILS(ahb_wr_rd_test);
    ahb_wr_rd_test(uvm::uvm_component_name name = "ahb_wr_rd_test") :
        uvm::uvm_test(name),
        top_env(0) {}

    virtual void build_phase(uvm::uvm_phase& phase)
    {
        std::cout << sc_core::sc_time_stamp() << " build_phase " << name() << std::endl;
        uvm_test::build_phase(phase);
        top_env = ahb_basic_env::type_id::create("top_env",this);
    }

    virtual void run_phase(uvm::uvm_phase& phase)
    {
        std::cout << sc_core::sc_time_stamp() << " started " << name() << std::endl;
        phase.raise_object(this);
        ahb_wr_rd_seq = new ahb_wr_rd_sequence("ahb_wr_rd_seq");
        ahb_wr_rd_seq->start(top_env->agent->ahb_sequencer_inst);
        phase.drop_object(this);
        std::cout << sc_core::sc_time_stamp() << " finished " << name() << std::endl;
    }
};
Create scv_extensions for the sequence item class i.e. for the transaction type

```cpp
SCV_EXTENSIONS(ahb_transaction) {
    public:
        scv_extensions<sc_uint<ahbConfig::AhbAddrWidth>> haddr;
        scv_extensions<sc_uint<ahbConfig::AhbDataWidth>[BURSTLENGTH]>
            hwdata;
        scv_extensions<sc_uint<ahbConfig::AhbBurstSize>> hburst;
        scv_extensions<sc_uint<ahbConfig::AhbSize>> hsize;

    SCV_EXTENSIONS_CTOR(ahb_transaction) {
        SCV_FIELD(haddr);
        SCV_FIELD(hburst);
        SCV_FIELD(hsize);
        SCV_FIELD(hwdata);
    }

    bool has_valid_extensions() { return true; }
};
```

Create constraints class using smart_ptr of sequence item class type

```cpp
class ahb_trans_constraints : virtual public scv_constraint_base {
    public:
        scv_smart_ptr<ahb_transaction> req;

    SCV_CONSTRAINT_CTOR(ahb_trans_constraints) {
        SCV_CONSTRAINT((req->haddr() * 0x3) == 0x0);
        SCV_CONSTRAINT((req->hburst() >= ahbConfig::HBURST_SINGLE) &&
            (req->hburst() <= ahbConfig::HBURST_INCR16));
        SCV_CONSTRAINT((req->hsize() >= ahbConfig::HSIZE_BYTE) &&
            (req->hsize() <= ahbConfig::HSIZE_WORD));

        // For wrapping bursts, start address from an address
        // other that 0x00 offset
        SCV_CONSTRAINT(
            if_then(req->hburst() == ahbConfig::HBURST_WRAP4,
            ((req->haddr() * 0x7) != 0x0) );
        }
};
```
Hierarchical Sequences using Random Sequence Items (SCV)

class ahb_wr_rd_sequence : public uvm::uvm_sequence<ahb_transaction>
{
    public:
        UVM_OBJECT_UTILS(ahb_wr_rd_sequence);
        UVM_DECLARE_P_SEQUENCER(ahb_sequencer<ahb_transaction>);
        ahb_if* ahb_vif_seq;
        ahb_wr_rd_sequence(const std::string& name = "ahb_wr_rd_sequence") :
            uvm::uvm_sequence<ahb_transaction>(name) {
            uint8_t xactType;
            unsigned addrValue;
            unsigned dataValue;
            void body()
            {
                UVM_INFO(this->get_name(), "Starting sequence", uvm::UVM_INFO);
                ahb_trans_constraints constr_req("constr_req");
                scv_smart_ptr<ahb_transaction> rand_smart_ptr_ahb_pkt;
                ahb_basic_sequence* ahb_seq;
                ahb_seq = new ahb_basic_sequence("ahb_seq");
                constr_req.next();
                rand_smart_ptr_ahb_pkt.write(constr_req.req.read());
                ahb_seq->xactType = rand_smart_ptr_ahb_pkt->hwrite;
                ahb_seq->hburstValue = rand_smart_ptr_ahb_pkt->hburst;
                ahb_seq->addrValue = rand_smart_ptr_ahb_pkt->haddr;
                ahb_seq->dataValue = 0xabababab;
                ahb_seq->start(m_sequencer);
                constr_req.next();
                rand_smart_ptr_ahb_pkt.write(constr_req.req.read());
                ahb_seq->xactType = rand_smart_ptr_ahb_pkt->hwrite;
                ahb_seq->hburstValue = rand_smart_ptr_ahb_pkt->hburst;
                ahb_seq->addrValue = rand_smart_ptr_ahb_pkt->haddr;
                ahb_seq->dataValue = 0xabababab;
                ahb_seq->start(m_sequencer);
                UVM_INFO(this->get_name(), "Finishing sequence", uvm::UVM_INFO);
            }
    }

class ahb_basic_sequence : public uvm::uvm_sequence<ahb_transaction>
{
    public:
        UVM_OBJECT_UTILS(ahb_basic_sequence);
        uint8_t xactType;
        unsigned addrValue, dataValue, hburstValue, hsizeValue;
        ahb_basic_sequence(const std::string& name = "ahb_basic_sequence") :
            uvm::uvm_sequence<ahb_transaction>(name) {
            void body()
            {
                UVM_INFO(this->get_name(), "Starting sequence ahb_basic_sequence", uvm::UVM_INFO);
                ahb_transaction* req_pkt;
                ahb_transaction* rsp;
                req_pkt = new ahb_transaction();
                rsp = new ahb_transaction();
                single_wr_rd(addrValue, xactType, dataValue, req_pkt, rsp);
            }
            void single_wr_rd(unsigned addrValue, unsigned xactType, unsigned dataValue,
                ahb_transaction* req_pkt, ahb_transaction* rsp)
            {
                UVM_INFO(this->get_name(), "Initiating non-burst accesses", uvm::UVM_INFO);
                req_pkt->haddr = addrValue;
                req_pkt->hsel = 1;
                req_pkt->hready = 1;
                req_pkt->htrans = ahbConfig::HTRANS_NONSEQ;
                req_pkt->hsize = hsizeValue;
                req_pkt->hwrite = xactType;
                req_pkt->hwdata[0] = (sc_uint<32>)dataValue;
                this->start_item(req_pkt);
                this->finish_item(req_pkt);
                this->get_response(rsp);
            }
    }
}
```cpp
class ahb_transaction : public uvm_randomized_sequence_item {
    public:
        UVM_OBJECT_UTILS(ahb_transaction);

    // define some rand variables
    crv_variable< sc_uint< ahbConfig::AhbAddrWidth > > haddr;
    crv_variable< sc_uint< ahbConfig::AhbSize > > hsize;
    crv_variable< sc_uint< ahbConfig::AhbDataWidth> > hwdata[BURSTLENGTH];
    crv_variable< unsigned > hburst;

    // Add some constraints
    crv_constraint valid_hburst_range {HBURST_SINGLE <= hburst() <= HBURST_INCR16};
    crv_constraint valid_hsize_range {HSIZE_BYTE <= hburst() <= HSIZE_WORD};
    crv_constraint valid_addr_range {haddr() * 0x3 == 0x0};
    crv_constraint addr_for_wrap_burst {if_then(hburst() == HBURST_WRAP4, (haddr() * 0x7) != 0x0)};

    // Constructor
    ahb_transaction(crv_object_name name = "ahb_transaction") : uvm_randomized_sequence_item(name) {
        ...,
    }
};
```
Hierarchical Sequences using Random Sequence Item(CRAVE)

```cpp
#include "ahb_basic_sequence.h"

class ahb_wr_rd_sequence : public uvm_randomized_sequence<ahb_transaction>
{
  public:
    UVM_OBJECT_UTILS(ahb_wr_rd_sequence);
    ahb_wr_rd_sequence( crave::crv_object_name name = "ahb_wr_rd_sequence") : uvm_randomized_sequence<ahb_transaction> (name )
    {
      cout << "Entered constructor of ahb_wr_rd_sequence " << endl;
    }

  void body()
  {
    UVM_INFO(this->get_name(), "Starting sequence", uvm::UVM_INFO);
    ahb_basic_sequence* ahb_seq;
    ahb_seq = new ahb_basic_sequence("ahb_seq");
    ahb_seq->hbustValue = ahbConfig::HBURST_SINGLE;
    ahb_seq->start(m_sequencer);
    UVM_INFO(this->get_name(), "Finishing sequence", uvm::UVM_INFO);
  }
};
```
Base Sequence With CRV_VARIABLE

class ahb_basic_sequence : public uvm_randomized_sequence<ahb_transaction>
{
    public:
        UVM_OBJECT_UTILS(ahb_basic_sequence);
        crv_variable<uint8_t  >  xactType;
        crv_variable<unsigned >  addrValue;
        crv_variable<unsigned >  dataValue;
        unsigned hburstValue, hsizeValue;
        unsigned readData;
        unsigned returnResp;
    ahb_basic_sequence( crave::crv_object_name name = "ahb_basic_sequence") :
        uvm_randomized_sequence<ahb_transaction> ( name )
    {
        {}
    }
        virtual ~ahb_basic_sequence() {}
    
    void body()
    {
        UVM_INFO(this->get_name(), "Starting sequence ahb_basic_sequence", uvm::UVM_INFO);
        ahb_transaction* req_pkt;
        ahb_transaction* rsp;
        single_wr_rd(addrValue, xactType, dataValue, req_pkt, rsp);
    }
}

/* Method for initiating single writes/reads to particular address with specific data */

void single_wr_rd(unsigned addrValue, unsigned xactType, unsigned dataValue, ahb_transaction* req_pkt, ahb_transaction* rsp)
{
    UVM_INFO(this->get_name(), "Initiating non-burst accesses", uvm::UVM_INFO);
    this->randomize();
    req_pkt->haddr = addrValue;
    req_pkt->hsel = 1;
    req_pkt->hready = 1;
    req_pkt->htrans = ahbConfig::HTRANS_NONSEQ;
    req_pkt->hsize = hsizeValue;
    req_pkt->hwrite = xactType;
    UVM_DO_WITH(req_pkt, req_pkt->haddr() == addrValue);
    UVM_INFO(this->get_name(), "Exiting non-burst accesses", uvm::UVM_INFO);
}

© Accellera Systems Initiative
Potential benefits of UVM-SystemC Methodology

- Lesser development time for testbench components
  - Base library provides analysis ports and callbacks
- Lower ramp time for new users adopting the IP
  - Testbench framework well known in verification circles
- Saving in testcoding time for IP validation at SoC level (using UVM-SV)
  - Language specific updates between SC and SV via simple script
- Reduced time in testbench components coding for IP interfaces at SoC level
  - Re-use of custom bus functional model written at IP level
- Overall less man hours for IP and SoC validation
  - Same owner can work on IP and SoC validation
- Many scenarios can be covered at SystemC level of the design as runtimes are considerably high for RTL simulations (if using HLS).
Sample Conversion Capabilities of the UVM-SC to UVM-SV script

• Changing class extension syntax
  – \texttt{class ahb\_transaction : public uvm\_randomized\_sequence\_item to class ahb\_transaction extends public uvm\_randomized\_sequence\_item}

• Updating the component phase arguments
  – \texttt{void run\_phase(uvm::uvm\_phase\& phase) to function void run\_phase(uvm::uvm\_phase phase)}

• Modifying the constructor calls
  – \texttt{ahb\_driver( uvm::uvm\_component\_name name = "ahb\_driver"):
    uvm::uvm\_driver<ahb\_transaction>( name ),ahb\_pipeline\_lock(1)
    { ... } to function new (string name = "ahb\_driver"):
    super.new(name);
    endfunction}

• Replacing loop constructor brackets with begin-end
  – \texttt{if(!uvm\_config\_db<ahb\_if*>::get(this, ",\\"\", "vif", ahb\_vif)) { ... } to
    if(!uvm\_config\_db<ahb\_if*>::get(this, ",\\"\", "vif", ahb\_vif)) begin ... end}
Conclusion

• UVM-SystemC based validation framework enables development of configurable, re-usable and structured components
• Standard implementation technique enables resilient testbench across multiple users
• This methodology should be adopted across companies and EDA vendors to make validation truly language agnostic and enhance the UVM-SystemC VIP portfolio
Thanks You!

Questions?