2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

UVM Made Language Agnostic – Introducing UVM For SystemC

Application of UVM-SystemC

Akhila M, Component Design Engineer, Intel, Bangalore, India (akhila.m@intel.com)

***Abstract*— the ability of SystemC to design concurrent processes using plain C++ libraries, has made it one of the prominent choices for System level modelling and High Level Synthesis (HLS). Even though SystemC has a dedicated verification library (SystemC Verification Library- SCV), there is no standardization in the testbench architecture and components to be used. This leaves loose ends in the validation capabilities and prevents easy reuse of testbench components across multiple users.**

This paper is a report on our findings using the UVM library for SystemC , which helps create a layered testbench in line with UVM standard for SystemVerilog. We will cover the details of all the major and must-have components for a UVM-SystemC based IP validation framework and also include ways to add constrained randomization using SCV. We also share some insights on the potential benefits w.r.t test and component re-use, when adopting UVM-SystemC methodology.

Keywords— uvm-systemc, verification framework, configurable testbench

I. INTRODUCTION

UVM-SystemC implements the Universal Verification Methodology (UVM) base classes in SystemC. This is developed with an intention of bringing the UVM capabilities to the Electronic System Level (ESL) world. The development is an ongoing effort in the Accellera workgroup with many important features already implemented. An alpha version of the library has been released in 2016. The use case explained in this paper is achievable using the current set of classes in the library.

One of the major applications for such a framework will be to validate an IP written in SystemC. In this paper we intend to share our experience of using UVM-SystemC for developing an IP validation framework, by building an AHB master bus functional model. This framework can also be reused at system or SoC level where the IP will be plugged-in. The source code might need some obvious modifications if the SoC uses UVM-SystemVerilog, which is still worth the reuse benefits of the tests and components.

The paper is divided into five sections. In section II we touch upon a few standard UVM techniques for component interaction and phasing. Section III gives a detailed explanation of each component in a standard UVM-SystemC environment, with code snippets. In section IV we share difference in randomization techniques when using SCV v/s CRAVE and also show some potential advantages of using UVM-SystemC for IP validation. Section V concludes the paper by stating the gain of using UVM-SC and the recommendations for new users.

II. PHASING AND HANDSHAKE MECHANISMS

A. Simulation phasing in UVM-SystemC

With the idea of bringing the UVM-SystemC phasing mechanism closer to UVM phases, the following implementation has been done to merge SystemC phases with UVM phases. The objection mechanism is supported to manage phase transitions. The UVM phases are broadly classified into three sections:

1. Pre-run Phase: This set includes build, connect, end_of_elaboration and start_of_simulation phases. Out of all these, the build phase is the only phase which is top-down as, in this phase, the connection of each component in the framework is identified and a virtual testbench structure is built. All components must have a build phase. In the connect phase, the analysis ports between various modules gets connected. The implementation of the other two phases is not mandatory and is application specific.

2. Run Phase: This is the only time consuming phase in the simulation. All SystemC processes should be executed in this phase. Run phases in all the components execute in parallel. UVM-SystemC also supports the optional sub-phases in the run phase.
3. Post-run Phase: This set is primarily responsible for doing any sanity checks and reporting at the end of simulation. User can add log parsing or data collation as part of these phases.

B. Handshake between UVM-SystemC components

UVM base library has provided a set of method calls for the sequence (running on a sequencer) and the driver to accomplish the handshaking between them. This is demonstrated in the figure 1 below:

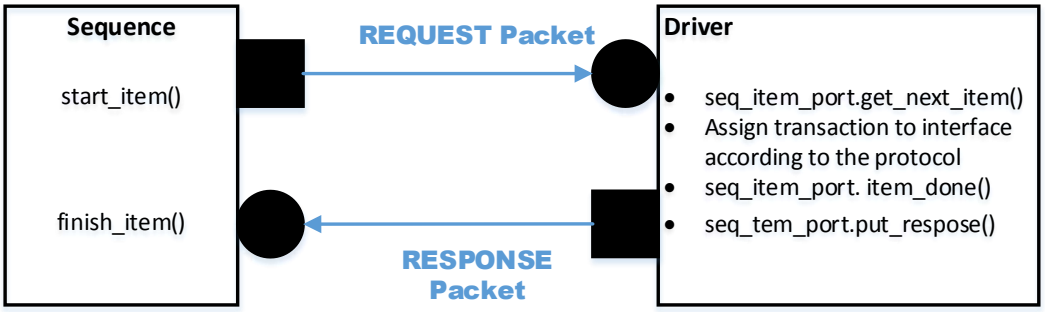

Figure 1: Handshake between sequence and driver

III. UVM-SystemC Testbench Architecture

In this section we share the architecture of an IP validation framework developed for an AHB slave interface. The testbench mimics the AHB master behavior. The structure of the components remains the same for any protocol, only the base implementation of some methods changes. This will be discussed in detail in the following sub sections.

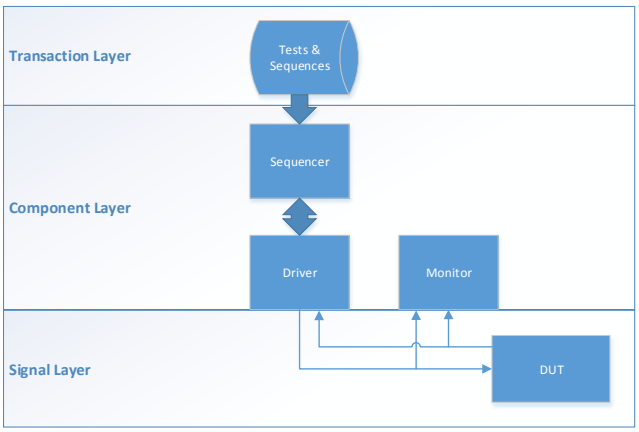

Figure 2. UVM-SC Layered Architecture

The UVM-SystemC methodology provides a layered architecture for the testbench components as seen in figure 2 above. These can be divided into three major categories: transaction layer, component layer and signal layer.

A. Transaction Layer

This layer is derived from the actual IP specification and determines what scenarios are to be presented to the Design Under Test (DUT) to achieve good feature coverage. The sequences are the core of the validation framework as they determine the actual "type" and "amount" of traffic to be pumped into the DUT.

```
class ahb_basic_sequence : public uvm::uvm_sequence<ahb_transaction>
{
    public:
        UVM_OBJECT_UTILS(ahb_basic_sequence);
        UVM_DECLARE_P_SEQUENCER(ahb_sequencer<ahb_transaction>);
        uint8_t xactType;
        unsigned addrValue;
        unsigned dataValue;
        unsigned hburstValue, hsizeValue;
        unsigned bursTdataValue[16];

        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();
            if(hburstValue == 0)
            {
                single_wr_rd(addrValue,xactType,dataValue, req_pkt, rsp);
            }
            else
            {
                burst_wr_rd_incr(addrValue, xactType, bursTdataValue, req_pkt, rsp,
                hburstValue);
            }
        }

        /* 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);
            req_pkt->haddr  = addrValue;
            req_pkt->hsel   = 1;
            req_pkt->htrans = ahbConfig::HTRANS_NONSEQ;
            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);
            if(rsp->response_status == tlm::TLM_INCOMPLETE_RESPONSE)
                UVM_WARNING(this->get_name(), "Reset assertion happened in between");
            UVM_INFO(this->get_name(), "Exiting non-burst accesses", uvm::UVM_INFO);
         }
```

The tests correspond to the feature being validated and each test can initiate one or multiple sequences. A sample test is shown in the snippet below. One important point to note is about raising an objection before a sequence is started and dropping it once it's finished. This is needed to make the sequence block simulate till its completion.

```
#include "ahb_wr_rd_sequence.h"
class ahb_wr_rd_test : public ahb_basic_test
{
    public:
        ahb_wr_rd_sequence* ahb_wr_rd_seq;
        UVM_COMPONENT_UTILS(ahb_wr_rd_test);
        ahb_wr_rd_test(uvm::uvm_component_name name = "ahb_wr_rd_test") : ahb_basic_test(name ) {}
        virtual void build_phase(uvm::uvm_phase& phase)
        {
          std::cout << sc_core::sc_time_stamp() << ":build_phase " << name() << std::endl;
          ahb_basic_test::build_phase(phase);
          ahb_wr_rd_seq = new ahb_wr_rd_sequence("ahb_wr_rd_seq");
        }
        virtual void run_phase(uvm::uvm_phase& phase)
        {
          std::cout << sc_core::sc_time_stamp() << ": UVM test started" << name() << std::endl;
          phase.raise_objection(this);
          ahb_wr_rd_seq->start(top_env->agent->ahb_sequencer_inst);
          phase.drop_objection(this);
          std::cout << sc_core::sc_time_stamp()<< ":UVM test finished" << name() << std:: endl;
        }
}
```

To be able to control the parameters and timing of multiple sub-level sequences, we can create virtual sequences and start them from the test, instead of the base sequence (as shown below). To enable randomization, we create SCV extensions for the base packet class and add constraints to them. This object can be used in the base or virtual sequence using smart pointers to randomize the packet fields, as shown below:

```
#include "ahb_basic_sequence.h"
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_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_pkt;
            ahb_basic_sequence* ahb_seq;
            ahb_seq = new ahb_basic_sequence("ahb_seq");
            constr_req.next();
            rand_smart_ptr_pkt.write(constr_req.req.read());
            ahb_seq->xactType = rand_smart_ptr_pkt->hwrite;
            ahb_seq->addrValue = rand_smart_ptr_pkt->haddr;
            ahb_seq->hburstValue = rand_smart_ptr_pkt->hburst;
            ahb_seq->dataValue = 0x12345678;
            ahb_seq->start(m_sequencer);
            constr_req.next();
            rand_smart_ptr_pkt.write(constr_req.req.read());
            ahb_seq->xactType = rand_smart_ptr_pkt->hwrite;
            ahb_seq->addrValue = rand_smart_ptr_pkt->haddr;
            ahb_seq->hburstValue = rand_smart_ptr_pkt->hburst;
            ahb_seq->start(m_sequencer);
         }
};
```

The constraints specification while using SCV requires creation of SCV extensions of the base transaction type, as shown in the snippet below. When using CRAVE, crv_variable and crv_constraints are specified to achieve the same thing.

```
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;}
};
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) ) ;
      );
    }
```

```
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) {
    };
};
```

B. Component Layer

The modules in this layer are derived from the uvm_component base class. The translation of sequences to DUT pin wiggles happens in this layer. The components in this layer are mapped to a particular "type" of object or transaction, on which they operate. The main modules in this layer are as follows:

- **Sequencer**: This component sets the arbitration for the sequences being run on it and sends the packets to the driver. The handshake between the sequencer and driver follows the command flow depicted in section II. The number and type of sequencers are directly related to the number and type of the interfaces the DUT has. For a multi interface DUT, there can be multiple sequencers instantiated in a class, which becomes the virtual sequencer. This is more common in an SoC testbench framework. The implementation of a sequencer can be found in [3].

- **Driver**: This is the major component, which interfaces with the DUT, and is responsible for driving traffic as per the specification. The driver pumps legal transactions into the DUT and collects the response from the DUT. One of the possible implementations of an AHB driver is shown below. Here the send_transaction method implements the AHB master behavior and is forked to mimic the pipelined nature of the AHB protocol. Please note, the complete implementation of the driver is not shown in the snippet.

- **Monitor**: This component, as the name suggests, is responsible for monitoring the requests going into the DUT and the response back from it. Checks are put in place to validate the protocol compliance along with some basic sanity checks. The sample code for the monitor is shown in the code below.

```
sc_time AHB_CLK(20, sc_core::SC_NS);

class ahb_driver : public uvm::uvm_driver<ahb_transaction>
{
  public:
    UVM_COMPONENT_PARAM_UTILS(ahb_driver);
    ahb_if* ahb_vif;
    sc_event* reset_event_driver;
    sc_semaphore ahb_pipeline_lock;
    // Constructor
    ahb_driver( uvm::uvm_component_name name = "ahb_driver"):
      uvm::uvm_driver<ahb_transaction>( name ),ahb_pipeline_lock(1)
      {
      std::cout << sc_core::sc_time_stamp() << ": constructor " << name << std::endl;
      }
    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<ahb_if*>::get(this, "*", "vif", ahb_vif)) {
        UVM_FATAL(this->get_name(), "AHB Virtual Interface not recieved in AHB driver");
      }
      if(!uvm_config_db<sc_event*>::get(this, "*", "reset_done", reset_event_driver)) {
        UVM_FATAL(this->get_name(), "Event not received after reset trigger");
      }
      UVM_INFO(this->get_name(), "build_phase Exited", UVM_LOW);
    }

    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_core::sc_spawn(sc_bind(&ahb_driver::send_transaction, this), "drive1"),
          sc_core::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()
    {
      UVM_INFO(this->get_name(), "In send_transaction method", UVM_LOW);
      ahb_transaction req, rsp;
      ahb_pipeline_lock.wait();
      UVM_INFO(this->get_name(), "In send_transaction method: before get_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);
      UVM_INFO(this->get_name(), "In send_transaction method: Got hready HIGH", UVM_LOW);
      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! Simulation aborted!");
  }
  void run_phase( uvm::uvm_phase& phase ) {
    ahb_transaction pkt;
    while (true) // monitor forever
    {
      std::ostringstream str;
      sc_core::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);
```

C. *Signal Layer*

In this layer everything translates to pin level transactions. The DUT virtual interface is passed dynamically across all testbench components and hence they can control (driver) or derive results (monitor) from the interface values.

```
int sc_main(int, char*[])
{
   ahb_if* dut_if_in = new ahb_if("dut_if_in");
   dut ahb_dut("ahb_dut");
   ahb_clk_reset_gen* clk_rst_gen = new ahb_clk_reset_gen("clk_rst_gen");
   dut_if_in->hclk(clk_rst_gen->ahb_clk);
   dut_if_in->hresetn(clk_rst_gen->reset_val);
   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;
}
```

IV. RESULTS

Using UVM-SystemC methodology we were able to create a configurable IP validation framework for our home grown IPs in SystemC. This helped in reusing the components from one IP to another, since the interface types are similar. As of now, there are very few other reports which clearly talk about the component handshake and the mandatory components for a validation framework using UVM-SystemC. This paper provides details of our implementations along with code snippets to help new users. The table 1 below showcases the difference in randomization approach when we chose SCV v/s CRAVE (an example of which can be seen in the snippet in section II A). The CRAVE methodology is explained in [5]. Table 2 showcases potential advantages and benefits w.r.t overall execution time, by adopting the UVM-SC methodology for IP validation.

Testbench Features	SCV	CRAVE
UVM-SC sequence item	Create SCV extensions for the basic sequence item class	Add crv_variables for the variables to be randomized
UVM-SC constraints	Add a separate class for constraint specification. Use smart_ptr bound to transaction type	Add crv_constraints in the same sequence item class
UVM-SC base sequence (with random variables)	Create object of constraint class and use smart_ptr to assign random values to base class variables. Use next() method of the constraint class.	Call randomize() method of the sequence item to generate random values for the crv_variables, in-line with crv_constraints.

Table 1: Comparison between SCV and CRAVE for the randomization technique/methods

Benefits of using UVM-SC framework	Remarks
Lesser time to code TB components i.e. driver, monitor etc. for IP validation	UVM-SC has advantage of having base libraries which provide handles like analysis ports and callbacks for faster implementation. Comparable time taken in both the methodologies, as the time is also determined by the interface complexity.
Less ramp time for new user adopting the IP(assumes user has required exposure to UVM and SystemC)	New user ought to understand the testbench framework, component interactions and test mechanisms in a non UVM-SC framework, as it's non-standard.
Saving in test coding time for IP validation at SoC level(SoC with UVM-SV framework)	UVM-SC can utilize a simple script to do language specific updates between SystemC and SystemVerilog, hence saving significant test coding time at full chip.
Reduced time in TB components coding for IP at SoC level	This is needed if a custom bus functional model written at IP level need to be reused at SoC. UVM-SC can utilize the script (mentioned above) along with some manual changes
Lesser man hours for IP and SoC validation	Same owner can work on IP and SoC validation, hence half the man hour (if it is an option!)

Table 2: Potential benefits of UVM-SC methodology as compared to a non-standard SystemC testbench

V. CONCLUSION

UVM-SC provides a way to develop configurable and reusable test suites and components for SystemC based validation. This paper shows the benefits of adopting a resilient and structured approach like UVM-SC for IP validation and recommends the use for better validated IPs. Using UVM-SC helped us in creating constrained random testcases, easily and hastened the eradication of functional bugs. The same environment was used to validate the SystemC cycle accurate code and RTL generated after HLS. This enabled end-to-end testing of the IP. We have showcased the major components/classes to be developed for implementing a validation framework using UVM-SystemC and have provided methods to support constrained randomization. We have also highlighted differences between SCV and the CRAVE approach w.r.t randomization and how the randomization implementations differ for end users.

REFERENCES

Papers [1], [2] and [3] provide an introduction to UVM-SystemC applications and give insights on few of the classes to be implemented in UVM-SystemC framework. More examples of using SCV can be found in [4]. The application of CRAVE in UVM-SC can be found in [5].

[1] Stephan Schulz, Thilo Vörtler, Martin Barnasconi, Karsten Einwich "UVM-SystemC Applications in the real world," DVCon Europe 2014

[2] Karsten Einwich, Thilo Vörtler, Martin Barnasconi , Thomas Kltoz , "Introducing the Universal Verification Methodlogy (UVM) in SystemC and SystemC-AMS" NASCUG 2014.

[3] Stephan Schulz, Thilo Vörtler, Martin Barnasconi, "UVM goes Universal – Introducing UVM in SystemC" DVCoN Europe 2015.

[4] David C. Black, Jack Donovan, Bill Bunton, Anna Keist, "SystemC: From The Ground Up", Springer second edition 2010.

[5] Stephan Gerth, Daniel Große "UVM goes random – Introducing CRAVE in UVM-SystemC" DVCon Europe 2016