Do not forget to 'Cover' your SystemC code with UVMC

Vishal Baskar
DVT Product Engineer
Siemens DISW
Vishal.Baskar@siemens.com

***Abstract*-We know how important coverage is in the verification cycle. But what if we are faced with a design that involves mixed language like System Verilog (SV) and SystemC (SC) with UVM Connect (UVMC)? Do we collect functional coverage from SC models? Or do we ignore them and use coverage from the SV side in a verification environment? Let us investigate how we make use of covergroups from SV using Transaction Level Modelling (TLM) from SC to SV using UVMC.**

I. INTRODUCTION

For decades, it is known that SystemC is used for modeling high levels of abstraction which can be used with SystemVerilog components. One of its greatest strengths is the versatility in mixed language designs offering various mechanisms and constructs required for embedded systems modeling. But when it comes to verification of those SystemC models, it is important to verify them at any level of abstraction, especially when it involves mixed languages. Today's modern system designs consist of multiple architectural components, involving a combination of hardware and software. SystemC enables a high-performance simulation of system behavior and creates accurate and efficient models of hardware-software components. Although the complexity of the systems can be handled using abstraction levels offered by C/C++/SystemC, verification of these is always a bottleneck.[1] SystemC is arguably becoming a standard and is widely being adopted by the industry for system-level modeling, hence it makes sense that the verification is extensively performed.

Unfortunately, when it comes to performing cover checks in the SystemC model directly, currently there is no standard. There have been attempts made by creating a functional coverage library for SystemC [2], and [3], to name a few. But what if the model utilizes UVM Connect [4] along with Transaction Level Modeling(TLM)-1.0 or 2.0 [5]? Further investigation and research are needed to ensure the IEEE-1647 e [6] functional coverage language features are supported in developing such libraries that would eventually become a standard in the future. Until then, in this paper, I have come up with a familiar solution of utilizing functional coverage usage by exporting them from SystemC to SystemVerilog each time a sampling event is called on the SystemC side. The remainder of the paper would discuss the Background of coverage analysis in Section II, and an illustration of the work in section III. Section IV would discuss the working of the approach using examples and sections V and VI would deal with the Conclusion of future work and references.

II. BACKGROUND

Coverage Analysis:

Functional Coverage is a metric used to measure how effectively the verification system or a design under test (DUT) has verified the system's functionalities. It is separate from other coverage metrics like code coverage, state machine coverage, assertion coverage, and so on. It is employed in all forms of verification. Functional coverage not only determines if a DUT is functioning properly but if you have done verifying the chip. [7]. As mentioned in the IEEE Std 1647, the process of measuring functional coverage involves three steps, as follows:

a. The user builds a coverage model, representing key architectural and micro-architectural features of the system being verified
b. Coverage data is collected during the simulation and accumulated for analysis
c. Coverage information is aggregated across many simulations and is analyzed to produce coverage scores

A coverage model includes one or more cover groups, which represent a data set to be sampled under certain conditions, typically with a sampling event. For each occurrence of the sampling event, each cover item samples one value and assigns that value into bins that the user has set. This approach is effective in creating a checklist to check, inserting them in the source code, and ensuring that every task is covered during verification.

Now the approach would be to send packets of data from the SC side to the SV side which can be used for sampling functional coverage each time the sample event is triggered. This way, verification of highly abstract components and system models can be done using tlm ports and sockets and the data can then be de-serialized on the SV side for functional coverage.

SystemC Modeling using UVMC:

This section would describe the essential aspects of SystemC modeling using UVMC. As we are aware, UVM Connect is an open-source UVM-based library that provides TLM-based connectivity and object passing between SystemC and SystemVerilog UVM models and components. UVM offers many TLM libraries such as ports, sockets, and interface ports. The libraries branch into 2 versions, TLM-1.0 and TLM-2.0. They contain functions to register ports and sockets of TLM models for cross-language connectivity, transaction switching between languages that support these connections, and allow SystemC to interact and control the UVM testbench

UVM Connect connects the SystemC model with UVM in a relatively straightforward process with the help of TLM sockets. The question arises when we are unsure if the transaction sent from the SC side has been verified, especially when packets of data have been sent from a high level of abstraction. Since SystemC along with the use of UVMC is becoming a common standard for design development, it is critical to verify the SC models to prevent crucial errors that trickle down to the system level.

III. DIAGRAM

Transactions in the form of packets are sent from the SC Side using TLM blocking ports vis UVM Connect. On the SV Side, as the transaction is received and unpacked, a sampling point is triggered by which we sample the coverage.

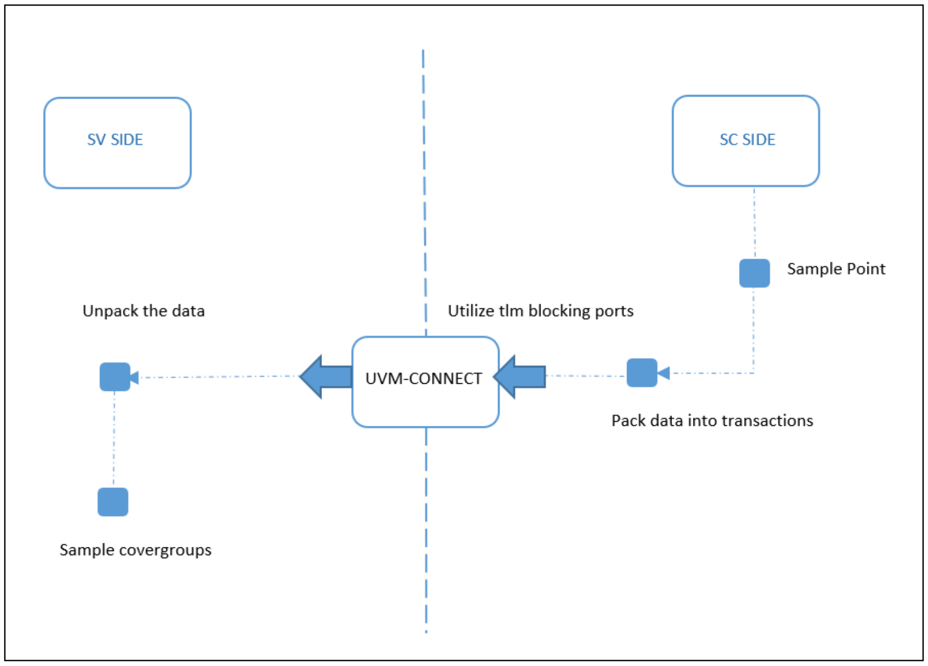


Figure: SC-SV connection

IV. WORKING

The connections are going to be simple. We have a SystemC model wherein we send packets of data and addresses transferred to the SV side via TLM-2.0 blocking transport. In this paper, we have two examples to import functional coverage to the SV environment from the SC side. Example 1 would deal with utilizing TLM sockets and generic payloads and example2 with transaction packets made from the template specialization class.

Example 1:

SC Side:

This example defines the producer_uvm class, which derives from our generic SC producer. In it, we spawn a dynamic objector thread that calls uvmc_raise_objection to UVM's run phase. This keeps the simulation alive on the SV side while the base producer in SC generates stimulus. When the base producer is finished generating stimulus, it will notify a done **sc_event**. This is the sampling point in our paper to perform coverage once it is sent to the SV side.

We pack that class that connects to the SV side by using a simple initiator socket, which is a derivative of the TLM Core class tlm_initiator socket. The simple socket does not require a module to inherit and implement the initiator socket interface methods. Instead, you only need to register the interfaces you implement. This is what makes these sockets simple, flexible, and convenient. Here we send a generic payload in the form of a transaction. It is nothing but a transaction type used in TLM 2.0. It has various data members (with protected type), enums, and methods available.

```
class producer : public sc_module {
  public:
simple_initiator_socket<producer> out; // uses tlm_gp //Original
tlm_analysis_port<tlm_generic_payload> ap; //Original
int num_trans;
sc_event done;
producer(sc_module_name nm) : out("out"), ap("ap"),   num_trans(2) {
SC_THREAD(run);
  }
  SC_HAS_PROCESS(producer);
void run() {
    tlm_generic_payload gp;
    char unsigned data[8];//Original
    gp.set_data_ptr(data); //Original
    sc_time delay;
    while (num_trans--) {

      gp.set_command(TLM_WRITE_COMMAND);
      gp.set_address(rand());
      delay = sc_time(10,SC_NS)
      int d_size;
      d_size = (rand() % 8) + 1; // between 1 and 8
      gp.set_data_length(d_size);

      for (int i=0; i < d_size; i++) {
        data[i] = rand();
      }
      cout << sc_time_stamp()
           << " [PRODUCER/GP/SEND] "
           << "cmd:" << gp.get_command()
           << " addr:" << hex << gp.get_address() << " data:{ ";
      for (int i=0;i<gp.get_data_length();i++)
        cout << hex << (int)(data[i]) << " ";
      cout << "}" << endl;
      out->b_transport(gp,delay); //Sampling point triggered
      ap.write(gp);
  }
```

The SC top level module registers the in port with UVMC using lookup strings.

```
#include "producer.h"
class producer_uvm : public producer {
  public:
  producer_uvm(sc_module_name nm) : producer(nm) {
    SC_THREAD(objector);
  }
  SC_HAS_PROCESS(producer_uvm);
  void objector() {
    uvmc_raise_objection("run");
    wait(done);
    uvmc_drop_objection("run");   }
int sc_main(int argc, char* argv[])
{
  producer_uvm prod("producer");
  uvmc_connect(prod.out,"42");
  sc_start(-1);
  return 0; }
```

SV Side:

This is a simple SV consumer TLM model that prints received transactions of type uvm_tlm_generic_payload. The class uvm_tlm_generic_payload has the following type and provides a transaction definition commonly used in memory-mapped bus-based systems. It's intended to be a general-purpose transaction class that lends itself to many applications. The class is derived from uvm_sequence_item which enables it to be generated in sequences and transported to drivers through sequencers. It consist of many data types like m_command, m_data[], m_byte_enable, m_response_status, etc.

The in-socket exports the tlm_blocking transport interface which is implemented and used by the consumer. We declare a covergroup sv_header and observe the values of the types passed, like command and data from the SC side.

```
class consumer extends uvm_component;
   uvm_tlm_b_target_socket #(consumer) in;
     uvm_tlm_gp t; //TLM Generic Payload on the receiver side.
    covergroup sv_header with function sample(uvm_tlm_gp t);
                    TYPE : coverpoint t .command {
                    ignore_bins nop = {0};
                    bins all = {[1:5]};  }
```

```
ADDR : coverpoint t.m_address {
                    bins b1 = {[0:'hfffff]};
                    bins b2 = {[20'hfffff:28'hfffffff]};
                    bins b3 = {[28'hfffffff:40'hffffffffff]};
                    bins max = {[40'hffffffffff:64'hffffffffffffffff]};
                }
```

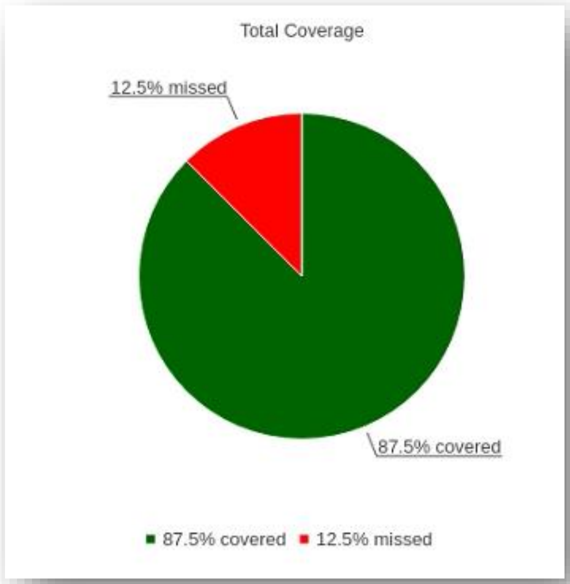
A virtual task b_transport receives the data of the generic payload using the analysis port. Just when it is received, we call it the coverage sampling function.

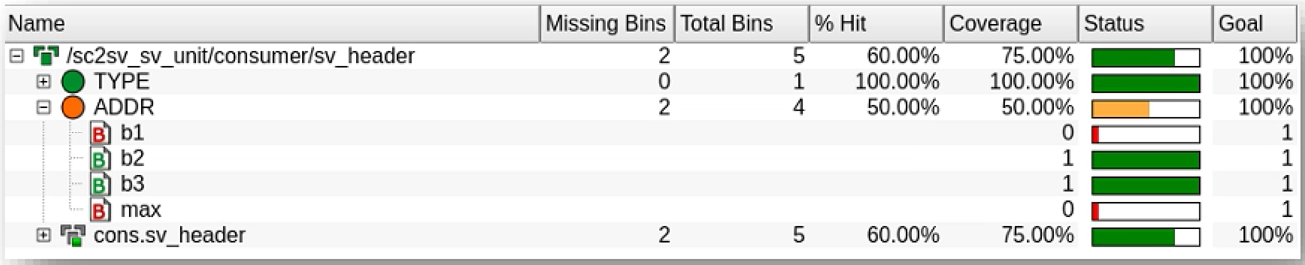
```
virtual task b_transport (uvm_tlm_gp t, uvm_tlm_time delay);
    `uvm_info("CONSUMER/PKT/RECV",{"\n",t.sprint()},UVM_MEDIUM)
    #(delay.get_realtime(1ns,1e-9));
    delay.reset();
    ap.write(t);
    sv_header.sample(t);
   endtask
 endclass
```

The sv_main top-level module creates and registers the consumer in target socket with UVMC. UVMC will connect this port with a port registered with the same lookup string and will match the producers in the port on the SC side.

```
//sv_main.sv
include "consumer.sv"
module sv_main;
  consumer cons = new("cons");

  initial begin
    uvmc_tlm #()::connect(cons.in, "42");
    run_test();
  end
```

Results:





Name	Missing Bins	Total Bins	% Hit	Coverage	Status	Goal
⊟ /sc2sv_sv_unit/consumer/sv_header	2	5	60.00%	75.00%		100%
⊞ ● TYPE	0	1	100.00%	100.00%		100%
⊟ ● ADDR	2	4	50.00%	50.00%		100%
b1				0		1
b2				1		1
b3				1		1
max				0		1
⊞ cons.sv_header	2	5	60.00%	75.00%		100%

This pie chart shows the percentage of coverage covered and missed. Two bins b1 and max have not been hit as the address is randomized and hence just 2 bins out of 4 were hit. This can be improved by using constraint randomization and cross-cover bins.

Example 2:

Here we use a converter class, which means the transaction definitions are allowed to be completely different i.e, the members (properties) of the transaction classes do not have to have the same declaration number, type, and order in both languages. The converter can adapt different transaction definitions and at the same time serialize the data. Functional coverage is applied when the transaction is sent as TLM transactions from SystemC to SystemVerilog. Instead of defining member functions of the transaction type to do the conversion, you should instead implement a template specialization of uvmc_converter<T>. This leaves conversion knowledge outside your transaction proper and allows you to define different conversion algorithms without requiring inheritance.

The default converter, uvmc_converter<T>, or any template specialization of that converter for a given packet type, is implicitly chosen by the C+ compiler. This means you will seldom need to explicitly specify the converter type when connecting via <uvmc_connect>.

SC Side:

On the SystemC side, we have the following class used as a transaction.

```
class packet
  {
    public:
    enum cmd_t { NOOP=0, READ, WRITE, RW };
    cmd_t cmd;
    unsigned int addr;
    vector<unsigned char> data;
    virtual void do_pack(uvmc_packer &packer) const {
      packer << cmd << addr << data;
    }
    virtual void do_unpack(uvmc_packer &packer) {
      packer >> cmd >> addr >> data;
}
```

In our case since we utilize the converter class' do_pack and do_unpack methods, one would stream members of your transaction to and from the <packer> variable. We do not need to define an external conversion class because its conversion is built into the transaction properly. The default converter will delegate to our transaction's ~do_pack~ and ~do_unpack~ methods. We do, however, define ~operator<< (ostream&)~ for our transaction type using <UVMC_PRINT> macros. With this, we can print the transaction contents to any output stream.

```
UVMC_PRINT_3(packet,cmd,addr,data);
```

The producer.cpp class contains the input transactions which are packed and sends it to the SV side. The producer sent TLM transactions in the form of blocking transport to SystemVerilog every time. A sampling event is reached when the packets are sent to the SV side.

```
//Producer.cpp
template <class T>
   class producer : public sc_module {
    public:
     sc_port<tlm_blocking_transport_if<T> > out;
     sc_event done;
     producer(sc_module_name nm) : out("out") {
       SC_THREAD(run);
     }
     SC_HAS_PROCESS(producer);
     void run() {
       sc_time delay;
       T t;
       int d_size;
       packet::cmd_t cmd;
       int unsigned addr;
       vector<char> data;
       char unsigned tmp;
       d_size = (rand() % 8) + 1;
       for(int i=0; i < d_size; i++) {
         tmp = rand();
         data.push_back(tmp);
       }
       t.set_data_copy(data);
       for (int i=0; i < d_size; i++) {
         delay = sc_time(10,SC_NS);
         cmd  = (packet::cmd_t)(rand()&3);
         addr = rand();
         t.cmd  = cmd;
         t.addr = addr;
         out->b_transport(t, delay); //Sampling point triggered
       }
```

Now let us look at the SV side. We have a simple converter class that we would receive from the SystemC side. And since we are using a converter class, the transaction classes need not be the same. Most transactions in SV should be defined this way as prescribed by UVM and it works with UVM Connect's standard SV converter. The `uvm_pack and `uvm_unpack macros expand into two or so lines of code that are more efficient than using the packer's API directly.

```
class packet extends uvm_sequence_item;

   `uvm_object_utils(packet)

   typedef enum { NOOP, READ, WRITE, RW } cmd_t;

   rand cmd_t cmd;
   rand int   addr;
   rand byte  data[$];
            .
            .
            .

virtual function void do_pack(uvm_packer packer);
     `uvm_pack_enum(cmd)
     `uvm_pack_int(addr)
     `uvm_pack_queue(data)
   endfunction

   virtual function void do_unpack(uvm_packer packer);
     `uvm_unpack_enum(cmd,cmd_t)
     `uvm_unpack_int(addr)
     `uvm_unpack_queue(data)
   endfunction
```

We have a consumer class that extends from packet_base. This is where we would apply coverage to the transaction received from the SystemC side. We can observe the values of the headers like the CMD and Address sent and we can define them in a covergroup.

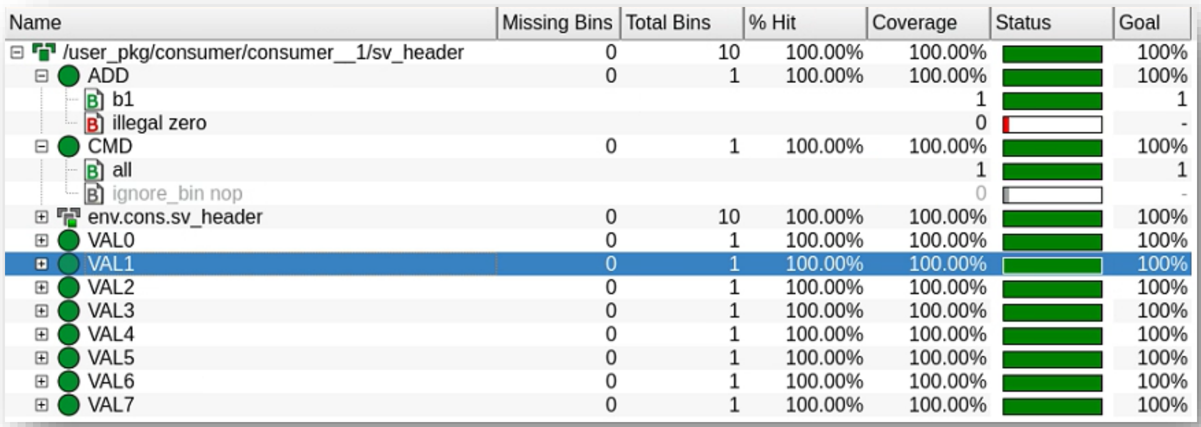
```
//consumer.sv
class consumer #(type T=int) extends uvm_component;
   uvm_tlm_b_target_socket #(consumer #(T), T) in;
…
   T t; //Template class transaction
```

```
    covergroup sv_header with function sample(T t);
        option.per_instance = 1;

        CMD : coverpoint t.cmd {
                ignore_bins nop = {0};
                bins all = {[1:3]};
             }

        ADD : coverpoint t.addr {
                illegal_bins zero = {0};
                bins b1 = {[0:32'hffffffff]};
                bins b2 = {[32'hffffffff:40'hffffffffff]};
                bins max = {[40'hffffffffff:64'hffffffffffffffff]};
            }

             VAL0 :  coverpoint t.data[0]{bins a0 = {[0:'hff]};}
             VAL1 :  coverpoint t.data[1]{bins a1 = {[0:'hff]};}
             VAL2 :  coverpoint t.data[2]{bins a2=  {[0:'hff]};}
             VAL3 :  coverpoint t.data[3]{bins a3=  {[0:'hff]};}
             VAL4 :  coverpoint t.data[4]{bins a4=  {[0:'hff]};}
             VAL5 :  coverpoint t.data[5]{bins a5=  {[0:'hff]};}
             VAL6 :  coverpoint t.data[6]{bins a6=  {[0:'hff]};}
             VAL7 :  coverpoint t.data[7]{bins a7=  {[0:'hff]};}
  endgroup
```

```
  virtual task b_transport (T t, uvm_tlm_time delay);
     `uvm_info("CONSUMER/PKT/RECV", $sformatf("SV consumer response:\n      %s", t.convert2string()),
UVM_MEDIUM)
     #(delay.get_realtime(1ns,1e-9));
     sv_header.sample(t);//Sample the coverage
    endtask
```

The SystemVerilog b_transport function is responsible for establishing a connection on the SV side and unpacking data. Once that happens, we would sample the coverage using sv_header.sample. A sampling event is triggered each time the b_transport from the SystemC side sends a TLM transaction and on the SystemVerilog side, the coverage function is sampled.

Results:



Name	Missing Bins	Total Bins	% Hit	Coverage	Status	Goal
/user_pkg/consumer/consumer__1/sv_header	0	10	100.00%	100.00%	🟩	100%
ADD	0	1	100.00%	100.00%	🟩	100%
b1				1	🟩	1
illegal zero				0	🟥	-
CMD	0	1	100.00%	100.00%	🟩	100%
all				1	🟩	1
ignore_bin nop				0	□	-
env.cons.sv_header	0	10	100.00%	100.00%	🟩	100%
VAL0	0	1	100.00%	100.00%	🟩	100%
VAL1	0	1	100.00%	100.00%	🟩	100%
VAL2	0	1	100.00%	100.00%	🟩	100%
VAL3	0	1	100.00%	100.00%	🟩	100%
VAL4	0	1	100.00%	100.00%	🟩	100%
VAL5	0	1	100.00%	100.00%	🟩	100%
VAL6	0	1	100.00%	100.00%	🟩	100%
VAL7	0	1	100.00%	100.00%	🟩	100%

V. Conclusion

In preparation for this approach, we attempted the use of tlm sockets and ports present in examples 1 and 2 respectively. In example 1, we utilized the uvm_tlm_generic_payload transaction which had data members like address, data, command, etc. Once we were able to identify the width and bins to cover, we were getting good results as seen in the results sections. In the second example, we used a template class <T> to send packets containing addresses and data to the SV side from the SC side. Instead of defining member functions of the transaction type to do the conversion, we implemented a template specialization class. We were able to capitalize on the do_pack and do_unpack methods of UVM and the coverage results show that all bins in the coverpoint were covered.

VI. Summary

In this paper, we attempted to create a novel approach to perform coverage on SystemC TLM transactions by exporting them to the SystemVerilog environment using two examples. One uses a generic payload and extracts coverpoint data from that and the other by using a packet that contains address, data, etc., and the other uses a custom template specialization class that consists of UVM's do_pack and do_unpack methods. We did discuss current methods and 3rd party libraries for performing functional coverage directly on SystemC Models, but none of them are industry-standard libraries. Currently, we do not have an industry-wide solution to perform coverage on SystemC models. Earlier work involved SystemC functional coverage library that was used to conduct a study on the CAN bus network.[9] There have been different attempts in the past to come up with a solution like Automatic Coverage

Analysis[10], and performing formal TLM property checking, but not functional coverage verification[11]. In the meantime, we can adapt to the approach attempted in this paper to discuss performing coverage at all levels as this has proven to be more flexible and easier to use.

Future Work:

Rather than having a class-based transaction passed as an object to a covergroup, utilizing constraint randomization and assertion on tlm_generic_payloads would certainly be more helpful. It would also help improve the overall coverage goal and to identify holes in coverage space. This can also be scaled up to signal processing models are designed in SC. It would be very beneficial to identify how various signals that are sent in, are sampled and to verify them using coverage, constraint randomization, etc. Any successful verification effort should aim to complete the target goals with the least amount of simulation cycles.

VII. References

[1]. A. Cimatti, A. Micheli, I. Narasamdya and M. Roveri, "Verifying SystemC: A software model checking approach," *Formal Methods in Computer Aided Design*, 2010, pp. 51-59.

[2]. Kuznik, Christoph and Wolfgang M̈uller. “Functional coverage-driven verification with SystemC on multiple level of abstraction.” (2011).

[3]. https://www.amiq.com/consulting/2018/02/22/cpp-implementation-of-functional-coverage-for-systemc/

[4]. Adam Erickson https://s3.amazonaws.com/verificationacademy-news/DVCon2013/Papers/MGC_DVCon_13_Transaction-Level_Friending_An_Open-Source_Standards-Based_Library.pdf extension://efaidnbmnnnibpcajpcglclefindmkaj/https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5554&rep=rep1&type=pdf

[5]. IEEE std 1666-2011 IEEE Standard for Standard SystemC Language Reference Manual

[6]. IEEE Standard for the Functional Verification Language e - IEEE Std 1647-2016 (Revision of IEEE Std 1647-2011)

[7]. ADVANCED VERIFICATION TECHNIQUES: A SystemC Based Approach for Successful Tapeout Leena Singh, Azanda Network Devices Leonard Drucker Cadence Design Systems Neyaz Khan Cadence Design Systems Inc.

[8]. UVM Cookbook - http://verificationacademy.com/uvm-ovm

[9]. C. Kuznik, G. B. Defo, and W. Muller, “Verification of a CAN bus model ̈ in SystemC with functional coverage,” in SIES 2010 Proceedings, 2010. [4] IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language, IEEE STD 1800-2009, pp. C1 –1285, 2009

[10]. Chen, Y., Du, X., Zhou, X., Peng, C. (2005). An Automatic Coverage Analysis for SystemC Using UML and Aspect-Oriented Technology. In: Shen, W., Lin, Z., Barthès, JP.A., Li, T. (eds) Computer Supported Cooperative Work in Design I. CSCWD 2004. Lecture Notes in Computer Science, vol 3168. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11568421_40

[11]. H. M. Le, D. Große and R. Drechsler, "Towards analyzing functional coverage in SystemC TLM property checking," 2010 IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010, pp. 67-74, doi: 10.1109/HLDVT.2010.5496658.

[12]. https://verificationacademy.com/verification-methodology-reference/uvmc-2.3.2/docs/html/files/examples/connections/sc2sv-cpp.html

[13]. https://www.amiq.com/consulting/2017/08/18/how-to-export-functional-coverage-from-systemc-to-systemverilog/