A SystemC-based UVM verification infrastructure

Mike Bartley, Test and Verification Solutions Ltd, Bristol, U.K (*mike@testandverification.com*)

Harshavardhan Narla, Test and Verification Solutions Ltd, Bangalore, India
(*harshavardhan.n@testandverification.com*)

***Abstract*—SystemC-based design is now gaining traction with improvements in the tools for implementation of such designs. TVS recently undertook a project to verify a number SystemC design IP blocks. It was decided to implement a SystemC test bench that would be UVM(Universal Verification Methodology)-compliant with a TLM2.0 interface. This entailed the following developments which are detailed within the paper:**

- **A C++ class library equivalent to the UVM class library.**
- **A Functional Coverage Library (FCL).**
- **Constraint based random verification was enabled through use of external randomization library called CRAVE.**

The above libraries are all freely available for engineers wanting to write UVM-based SystemC test benches. In this we describe these libraries in detail and explain the results of using them to verify SystemC design IP blocks.

Keywords—SystemC, UVM, UCIS, Requirements Driven Verification

I. THE TVM INFRASTRUCTURE

The first infrastructure developed was the C++ class library named TVM (TVS Verification Methodology) equivalent to the UVM (Universal Verification Methodology) class library. Tests can be compiled using the free GCC based compilers. Figure 1 below shows the structure of the TVM library.

- TVS implemented a library of base methodology classes and functions in C++ (including factory constructs, agents, monitors, scoreboard, drivers and sequencers) enabling easy conversion from TVM->UVM or UVM->TVM.
- The TVM library mimicked UVM phases such as compile, build, run, check etc.
- The library includes base classes for constraint driven randomization using CRAVE (Constrained RAndom Verification Environment). See section 2 below for more details.
- Functional coverage is enabled through aTVM Functional Coverage Library (FCL).
- Code Coverage is also possible through gcov and lcov tools.

Additional UVM similarities are detailed in the "UVM correspondence and compliance" section.

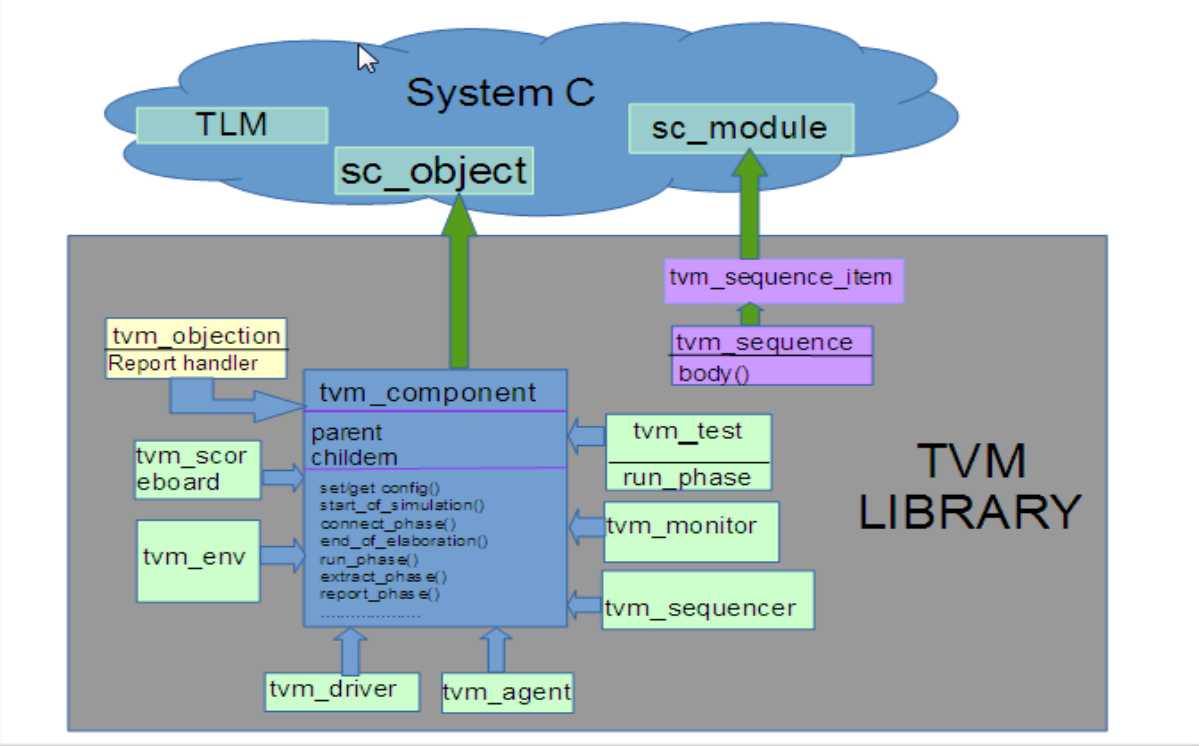

Figure 1:TVM Library structure

II. Infrastructure for Constraint Driven Random Verification

Constraint based random verification is enabled through use of external randomization library called CRAVE. The syntax of CRAVE has been designed to naturally fit with C++ and SystemC.

- The CRAVE library allows users to set constraints in a manner similar to UVM & System Verilog.
- CRAVE allows inline constraints, which can be formulated and changed incrementally at run-time.
- CRAVE is freely available under the MIT license. CRAVE is available via public repositories at GitHub.

III. Coverage through TVS Functional Coverage Library & asureSIGN

TVS developed a Functional Coverage Library(FCL) to enable Coverage Driven Verification(CDV). This library can be used either with TVM or individually on C++ based environments. The main features of the FCL are:

- The functional coverage report is generated in xml format using the "Tiny xml" library add-on. This format is compatible with TVS asureSign™ so that the coverage can be viewed against requirements.
- The FCL supports: multiple cover groups in a single instance; conditional coverage; exclusion or Inclusion of cross bins (by specific cover bin scope or bin specific); transition Bins (one series of incrementing transition); auto bins (up to 16 bits); cross coverage (for up to 4 cover points).

IV. UVM correspondence and compliance

The following features give a strong correspondence between TVM and UVM:

- Constructs for agents, monitors, drivers, sequencers and scoreboard are available.
- TVS has ensured that the factory registration and overriding concepts of UVM methodology are retained.

- Randomization is controlled using constraints similar to UVM / System Verilog via CRAVE.
- Ability to raise and drop objections.
- The config_db mechanism similar to UVM is available.
- Dynamic casting using DCAST similar to System Verilog's $cast.
- Support for time out mechanism as in UVM.
- Reporting severity similar to UVM (through the SystemC inbuilt support).
- Uses TLM ports (interface method class), similar to UVM.

The following System Verilog features are also supported:

- Verbosity control of debug messages.
- Fine process control ((like fork-join, fork-join none, fork-join_any, tracking process using process handle) (by using systemC's library, has inbuilt support).
- Interface class.

This enables reuse with respect toVerilog and System Verilog.

- The TVM code is easy to port to UVM and Vice-Verse.
- TVM can be re-used to verify RTL or for co-simulation between RTL and SystemC.

V. A VERIFICATION METHODOLOGY FOR TVM

The above infrastructure described in the previous sections meant that TVS was able to follow its standard verification flow:

- This starts with a feature extraction. For each block to be verified the interfaces and block functionality is analyzed to identify the major features.
- The next step is to identify how to verify each feature. Each feature can be mapped to a wide variety of verification goals such as:
    - functional coverage targets;
    - checkers such as scoreboards and assertions;
    - tests (constrained random with or without a seed, or directed);
    - code coverage targets;
    - properties to be proved via formal verification (again, not relevant to this project);
    - software execution;
    - silicon validation lab tests;
- A TVM test bench can now be developed and the verification goals mapped to test bench components. For example:
    - A functional coverage goal can be mapped to the TVM functional coverage points that implement them.
    - A test can be mapped to the TVM test thatimplements that goal.

The above verification methodology was supported via a Requirements Management tool asureSIGN. The tool was used to capture the feature extraction, verification goals and test bench elements, and the mappings between them. As tests are run they automatically report their results to asureSIGN via a UCIS(Unified Coverage Interoperability Standard)-compliant API (Application Programming Interface) and the code and functional coverage results are collected and automatically fed to asureSIGN via XML (EXtensible Markup Language) (based on the UCIS standard). For each regression asureSIGN automatically records the results (and the version control tag) so that progress can be monitored.

VI. SAMPLE TVM SETUP & CODE EXAMPLES

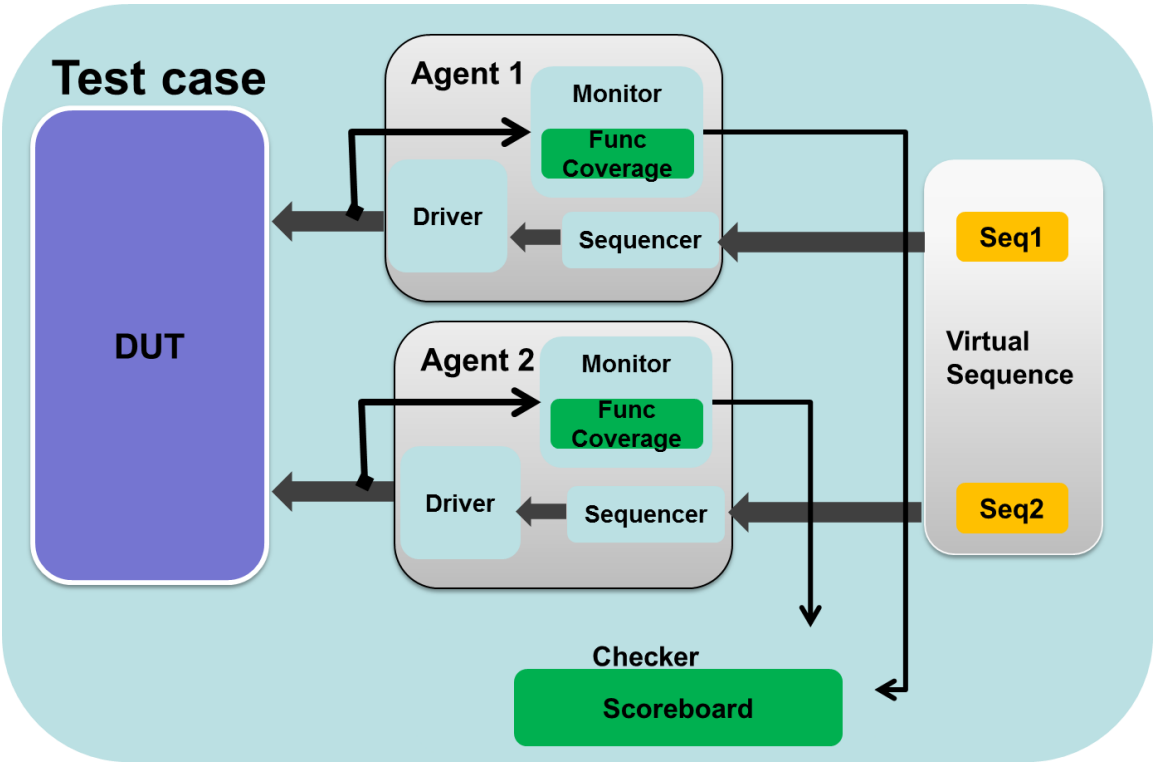

Figure 2: A Sample TVM TB setup

```
class demo_demo_top : public tvm_top_base {
   TVM_COMPONENT_UTILS(demo_demo_top);
  public :
            demo_demo_top(const sc_module_name& _name):tvm_top_base( _name)   {}
            ~demo_demo_top() {}
           protected :
             virtual void dump_trace();
             virtual void create_dut();
             virtual void connect_dut_and_tb();
 };
```

Note:

- For TVM methodology the C++ constructor acts as the build phase.
- "//..." in code indicates excess code removed which is needed for operability but not necessary in the explanation.

```
Class tvm_testcase : public tvm_test {
   TVM_COMPONENT_UTILS(tvm_testcase);
```

```
        //declare variables etc
        //.....................
          public :
                    demo_master_interface *mstr_intf;
                    demo_slave_interface  *slv_intf;
                    demo_env *env;
                    demo_virtual_sqr *v_sqr;


          tvm_testcase( const sc_module_name& _name ) : tvm_test( _name ), clk("system_clk",10,SC_NS)
        {
                    v_sqr = DCAST<demo_virtual_sqr*>(demo_virtual_sqr::type_id::create("demo_v_sqr"));
                    //connection between testbench components
                    env = DCAST<demo_env*>( demo_env::type_id::create("demo_env"));
                    mstr_intf = DCAST<demo_master_interface*> ( demo_master_interface::type_id::create(
          "demo_master_interface") );
                    slv_intf = DCAST<demo_slave_interface*> ( demo_slave_interface::type_id::create(
          "demo_slave_interface") );
                    tvm_config_db<demo_slave_interface>::set(this,
          "demo_env.demo_slave_agent.demo_slave_driver", "intf", slv_intf);
                    tvm_config_db<int>::set_config_int(this,"demo_env.demo_slave_agent.demo_slave_driver",
          "to_check", to_chk);
                    tvm_config_db<std::string>::set_config_string(this,"*","MSTR",to_CHK);
      }
      protected :
                    //systemc inbuilt phases
                    virtual void end_of_elaboration();
                    virtual void start_of_simulation();
                   //TVM methodlogy phases
                   virtual void connect_phase();
                   virtual void run_phase();
                   virtual void extract_phase();
                   virtual void check_phase();
                   virtual void report_phase();
  }


    void tvm_testcase::connect_phase() {
                  std::cout << "Demo_env connect Phase is called " << std::endl;
```

```
                mstr_intf->addr(addr);
                slv_intf->addr(addr);
                mstr_intf->wdata(wdata);
                slv_intf->wdata(wdata);
               //.............
                mstr_intf->clk(clk);
                slv_intf->clk(clk);

                //mstr and slave i/f connection needs to be done at top if DUT connections are needed.
                //We are doing this here since this setup has Mstr & slave back2back
                mstr_agnt->mstr->intf = mstr_intf;
                slv_agnt->slv->intf = slv_intf;
                //vsqr connection
                v_sqr->mstr_sqr = mstr_agnt->sqr;
                v_sqr->slv_sqr = slv_agnt->sqr;
}
```

```
class demo_env : public tvm_env {
   TVM_COMPONENT_UTILS(demo_env);
   //......
   public :
                demo_master_agent *mstr_agnt;
                demo_slave_agent *slv_agnt;
                demo_env(const sc_module_name& _name):tvm_env( _name)
                {
                        mstr_agnt  =  DCAST<demo_master_agent*>(  demo_master_agent::type_id::create(
                  "demo_master_agent") );
                        slv_agnt                                                                    =
                  DCAST<demo_slave_agent*>(demo_slave_agent::type_id::create("demo_slave_agent"));
                 }
   private:
                demo_env( const demo_env& _item);
                demo_env& operator = ( const demo_env & _item );
   //.......
}
```

```
class demo_master_agent : public tvm_agent {
  TVM_COMPONENT_UTILS(demo_master_agent);

  public :
        demo_master_driver *mstr;
        demo_master_sqr *sqr;
        demo_master_monitor *mon;

        demo_master_agent(const sc_module_name& _name):tvm_agent( _name)
        {
            mstr = DCAST<demo_master_driver*>( demo_master_driver::type_id::create(
          "demo_master_driver") );
            sqr = DCAST<demo_master_sqr*>( demo_master_sqr::type_id::create(
          "demo_master_sqr") );
            mon = DCAST<demo_master_monitor*>( demo_master_monitor::type_id::create(
          "demo_master_mon") );
        }

  private:
        demo_master_agent( const demo_master_agent& _item){}
        demo_master_agent& operator = ( const demo_master_agent & _item ){
            return (*this);
        }

  protected :
        //....
        virtual void connect_phase();
        demo_master_agent(){}
        virtual ~demo_master_agent(){}
//.......
};

void demo_master_agent::connect_phase() {
        mstr->sqr_port(*(sqr->exp));
}
```

```
class demo_master_driver : public tvm_driver {
```

```
TVM_COMPONENT_UTILS(demo_master_driver);
 public :
            Phase *phs;
            demo_master_interface *intf;
            sc_port < tvm_sqr_if<demo_reg_master_sequence_item> > sqr_port;
            demo_master_driver( const sc_module_name& _name ) :tvm_driver( _name )
                        ,sqr_port("mst_sqr_port")
            {
             phs=Phase::Instance();
            }
 protected :
           //.......
           virtual void run_phase();
 };

void demo_master_driver::run_phase() {
           demo_reg_master_sequence_item h;
           std::string to_CHK;
           tvm_config_db<std::string>::get_config_string(this,"MSTR",to_CHK);
           std::cout << "Master Run Phase called " << std::endl;
           while(1) {
                       sqr_port->get_next_item(h);
                       phs->raise_objection(this,"raise objection from Master Driver .... ");
                       intf->addr.write(int(h.addr));
                      //......
                       wait(intf->clk.posedge_event() );
                       while( intf->ack.read() == 0) {
                         wait(intf->clk.posedge_event() );
                       }
                      //......
                       sqr_port->item_done(h);
                       phs->drop_objection(this,"drop objection from Master Driver .... ");
           }
 }
```

```
class demo_master_sqr : public tvm_sequencer<demo_reg_master_sequence_item>
```

```
{
  TVM_COMPONENT_UTILS(demo_master_sqr);
  demo_master_sqr(const sc_module_name& _name): tvm_sequencer <demo_reg_master_sequence_item> ( _name ) {}
};
TVM_COMPONENT_REGISTER(demo_master_sqr);
```

```
class demo_master_sequence : public tvm_sequence <demo_reg_master_sequence_item> {
  TVM_COMPONENT_UTILS(demo_master_sequence);
  public :
        int no_of_trans;
        Phase *phs;
        demo_master_sequence(const char* _name) :tvm_sequence <demo_reg_master_sequence_item> (_name)
        {
              phs=Phase::Instance();
        }
        virtual void body();
      //.....
};
//.....

void demo_master_sequence::body() {
        phs->raise_objection(this,"raise objection from master sequence .... ");
        int i;
        demo_reg_master_sequence_item k;
        for( i = 0 ; i < no_of_trans; i++ ) {
              k.next(); //randomization done here through CRAVE
              k.wen = 1;
              k.valid = 1;
              start_item(k);
              finish_item(k);
        }
      phs->drop_objection(this,"drop objection from master sequence .... ");
}
```

```
using crave::rand_obj;
using crave::randv;
class demo_reg_master_sequence_item : public tvm_sequence_item, public rand_obj {
 TVM_COMPONENT_UTILS(demo_reg_master_sequence_item);
 public :
        randv< unsigned int > addr;
        randv< unsigned int > rdata;
        randv< unsigned int > wdata;
        bool wen;
        bool valid;

 demo_reg_master_sequence_item(rand_obj* parent = 0): rand_obj(parent), addr(this), rdata(this), wdata(this){
        constraint(addr() >= 0x00 && addr() <= 0xFF);
        constraint(wdata() <= 0xFFFF);
  }

 ~demo_reg_master_sequence_item() {}
 demo_reg_master_sequence_item& operator=(const demo_reg_master_sequence_item & _item);
};

demo _reg_master_sequence_item& bw_reg_master_sequence_item::operator=(const demo_reg_master_sequence_item & _item) {
        if( this == &_item) {
                return(*this);
        }
        this->addr = _item.addr;
        this->wdata = _item.wdata;
        this->rdata = _item.rdata;
        this->wen = _item.wen;
        this->valid = _item.valid;
        return(*this);
}
TVM_COMPONENT_REGISTER(demo_reg_master_sequence_item);
```

VII. SAMPLE TVM FUNCTIONAL COVERAGE EXAMPLE

Below is a sample implementation of the TVM functional Coverage Library in a C++ code.

```
#include "tvm_func_cov.h"

COVER_GROUP_START(CG1)
  int ph_addr;
  int ph_data;

 COVER_POINT_ST(ADDR)
  EX_AUTO_BINS(addr)
 COVER_POINT_END

 COVER_POINT_ST(DATA)
  AUTO_BINS(dataautobin)
 COVER_POINT_END

 ADD_COVER_POINT
  COVER_POINT(ADDR,ph_addr,2)
  COVER_POINT(DATA,ph_data,5)
 END_COVER_POINT

 SAMPLE_COVER_POINT
  SAMPLE(ADDR)
  SAMPLE(DATA)
 END_SAMPLE_COVER_POINT

COVER_GROUP_END


  int main() {
  CG1 obj("CG1");
  obj.set_db_name("file1.addingcp");
```

```
    obj.add_cover_point();
    obj.ph_addr = 2;
    obj.sample();
    obj.ph_data = 666;
    obj.sample();
    obj.display();

    return 0;
  }
```

VIII. INITIAL DEPLOYMENT

The above methodology has been applied to block verification with the following initial results:

- Re-usable TVM agents were developed for the proprietary protocols based internal busses. These were re-used on subsequent IP blocks.
- High rates of both functional and coverage were achieved on all blocks.
    - 100% statement coverage on all blocks.
    - The structured feature extraction process was converted to a functional coverage model. TVS were then able to generate tests to hit 100% of the plan.
    - Throughout the coverage closure process progress was tracked via asureSIGN and reports generated.
- 21 bugs were discovered.

IX. FUTURE TVM DEVELOPMENTS

The following TVM improvements have been identified:

- Call-backs are not implemented. For now, as it is pure C++ library, C++ call-backs can be used (Courtesy: tedfelix.com/software/C++-callbacks.html).
- TVS has a Virtual Sequencer based mechanism but it needs further improvement to make it more compatible with the p_sequencer / m_sequencer based methods in UVM.
- There is currently no UVM-RAL model equivalent.
- uvm_event, uvm_event_pool and message passing using these functions need to be added.
- The functional coverage library support needs to be extended.
- The SystemC assertion library needs to be tested.

X. CONCLUSION

The first Infrastructure developed was a C++ class library (named TVM) equivalent to the UVM class library. Constraint based random verification was then enabled through the use of an external randomisation library called CRAVE. A Functional Coverage Library was also developed to enable Coverage Driven Verification (CDV). This library can be used either with TVM or individually on C++ based environments.

The application of the TVS verification methodology was successful. Constraint-based random verification was enacted and high coverage rates were achieved for both functional and code coverage (the latter enabled by freely available gcov and lcov tools). This enabled us to achievehigh rates of bug detection.

The SystemC-UVM infrastructure developed by TVS is now freely available for other engineers.

XI. REFERENCES

[1] Hoang M. Le and Rolf Drechsler, "CRAVE 2.0: The Next Generation Constrained Random Stimuli Generator for SystemC," DVCON Europe, 2014
[2] Universal Verification Methodology (UVM) 1.1 User's Guide, Accellera, May, 2011
[3] Universal Verification Methodology (UVM) 1.2 Class Reference, Accellera, Jun, 2014