Methodology for checking UVM VIPs

Milan Vlahovic, Veriest Solutions, Belgrade, Serbia
Ilija Dimitrijevic, Veriest Solutions, Belgrade, Serbia

Introduction

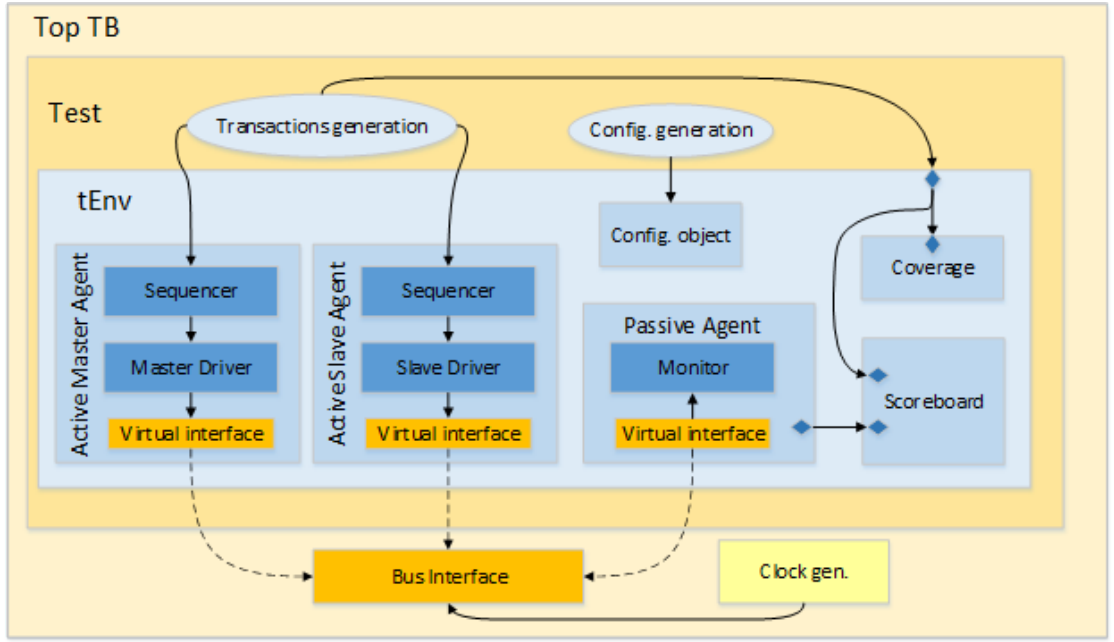
- If the VIPs are tested thoroughly by the developers?
- Debugging VIPs by user should be avoided
  - It is tricky to debug code written by someone else
  - VIP's files can be encrypted
  - User isn't protocol expert
- VIPs should be proven and bugs free components when used in verification environments
- Comprehensive testing is necessary during development

What to check?

- All main VIP's functions
  - Driving transactions
  - Monitoring interface, collecting and reporting transactions
  - Checking protocol rules (assertions and procedural checkers)
  - Reset

Test environment architecture

- Self-checking test environment for all the functions
- Active master and slave agents
  - Checking master and slave drivers
  - Connected via the same Bus interface
- One passive agent
  - Checking monitor
- Test
  - Generates transactions and configuration
  - Sends transactions directly to sequencers
- Scoreboard (SB)
  - Compares transactions from monitor against transactions from the test
- Coverage



Driving and monitoring check

- Tests create all legal master & slave transactions and send them to sequencers
- SB checks drivers and monitor correctness
  - Fails if drivers don't drive correctly
  - Fails if monitor doesn't collect generated transactions
  - Fails if monitor collects wrong transaction
- Master commands transactions
  - Configure slave agent to not block master commands
- Slave responses transactions
  - Checking ready/busy signal driving & monitoring in case of receiver
  - Checking automatic responses of responder
  - Checking memory model of responder
  - Checking all types of responder's responses

Checking protocol checkers

- Error injection
- Use SB from tEnv instead of visual inspection
- Extended transaction – enumeration with values for all error types
- Extended driver – injecting errors
- Extended monitor – checking the checkers
  - Procedural checkers – implemented in monitor
  - Assertion checkers – implemented in interface
  - Checkers implementation is slightly modified for self-checking
- Specific test for checking the checkers

Extended transaction with error types

- Enumeration value for every type of protocol violation errors

```
// Error types enumeration type
typedef enum {NO_ERR,
              // procedural checkers errors
              INACT_MIN_WR,
              INACT_MIN_RD,
              WR_MIN,
              OE_MIN,
              // intf. assertions errors
              NO_STBL_DATA_DUR_WE,
              NO_STBL_ADDR_DUR_WE,
              NO_STBL_WAIT_DUR_WE,
              OE_WE_ACT_SAME_TIME,
              NO_STBL_CS_DUR_WE,
              NO_STBL_ADDR_DUR_OE,
              NO_STBL_CS_DUR_OE} error_type_t;

class async_par_bus_err_trans#(int DATA_WIDTH = 16) extends async_par_bus_transaction#(DATA_WIDTH);
  `uvm_object_param_utils(async_par_bus_err_trans#(DATA_WIDTH))

  rand error_type_t  error_type; // Type of protocol error
  constraint error_type_c {soft error_type == NO_ERR;}
```

- Enumeration value as an error ID

Extended driver injecting errors

- Main task in extended driver

```
task async_par_bus_err_inj_mst_drv::drive_if(async_par_bus_transaction#(DATA_WIDTH) tr);
   async_par_bus_err_trans#(DATA_WIDTH) err_tr;
   if(!$cast(err_tr, tr))
      `uvm_fatal(get_name(), "Error transaction cast failed!");
   if(err_tr.error_type != NO_ERR)
      inject_err(err_tr);
   else
     super.drive_if(err_tr);
endtask: drive_if
```

- Error injection example
  - Using the code of basic driver as much as it is possible

```
task async_par_bus_err_inj_mst_drv::inject_err(async_par_bus_err_trans#(DATA_WIDTH) tr);
   `uvm_info(get_name(), $sformatf("Driving error type %0s.", tr.error_type.name), UVM_LOW)

   case (tr.error_type)
      NO_STBL_DATA_DUR_WE: begin
         int data_change_time;
         bit [DATA_WIDTH-1:0] new_data;
         tr.command = WRITE;
         fork
            super.drive_if(tr); // Drive legal Write command
            begin
               wait (vif.WE == 0);
               std::randomize(data_change_time) with {data_change_time > 0; data_change_time < tr.twr;};
               std::randomize(new_data) with {new_data != tr.data_in;};
               // Change the data before the command is finished
               #(data_change_time*1ns);
               vif.DATA_REG = new_data;
            end
         join
      end
```

Checking procedural checker

- Error processing function call instead of immediate report

```
@(posedge vif.WE);
//Timing checker for write strobe time - the minimum time that the WE signal needs to be asserted active
if($time-time_check < `WRITE_STROBE_MIN)
    process_error($sformatf( "WR command timing eror: WE signal was active for %0tns, but minimum is %0tns",
                             $time-time_check,`WRITE_STROBE_MIN), 3);
```

```
function void async_par_bus_monitor::process_error(string err_msg, int err_id);
    if(check_en)
        `uvm_error(get_name(), err_msg)
    err_flag = 1;
endfunction
```

- Two arguments of the function
  - Error message string to be printed
  - Procedural error ID – matching enumeration value of the error type
- Function prints error by default

Checking assertion checkers

- Error flag and self-check control bit in SV interface – 0s by default
- Implement *else branch* of the assertion

```
//ASSERTION: Stable data during Write operation
property stable_data_during_WE;
    @(DATA)
    disable iff (!MR || !assertion_checkers_enabled)
    !CS |-> WE;
endproperty
assert property (stable_data_during_WE)
    `process_assertion("Data bus NOT stable during active WE!", 1);
```

```
`define process_assertion(MSG, ERR_ID) \
  else begin \
    if (!self_check_on) \
      $error(MSG); \
    assert_error = ERR_ID; \
  end
```

- Self-check enabled -> error report is overridden with setting error flag to assertion error ID
- Assertion error ID - matching enumeration value of the error type

Extended monitor

- Enables checking of all the protocol checkers
- Function observing error flag from interface
  - Waits for interface error
  - Calls error processing function
- Error processing function
  - Creates transaction of error type according to given error ID
  - Writes error transaction to analysis port – sends to SB

```
task async_par_bus_err_check_mon::run_phase(uvm_phase phase);
    fork
        super.run_phase(phase);
        wait_assertion();
    join
endtask: run_phase

task async_par_bus_err_check_mon::wait_assertion();
    forever begin
        wait(vif.assert_error != 0);
        process_error("assertion error", vif.assert_error+OE_MIN);
        wait(vif.assert_error == 0);
    end
endtask
```

```
function void async_par_bus_err_check_mon::process_error(string err_msg, int err_id);
    string error_src;
    async_par_bus_err_trans err_tr = new();
    err_tr.error_type = err_id;
    if(vif.assert_error)
        error_src = {err_msg, " of type ", err_tr.error_type.name};
    else begin
        error_src = {"procedural checker error: ", err_msg};
        err_flag = 1;
    end
    `uvm_info(get_name(), $sformatf("Monitor detected %s.", error_src), UVM_LOW);
    an_port.write(err_tr);
endfunction
```

Test checking the checkers

- Basic classes are overridden by new types supporting error injection
  - Transaction -> extended transaction with errors enumeration
  - Drivers -> extended drivers injecting the errors
  - Monitor -> extended monitor processing the errors

```
class checkers_test extends base_test;
   `uvm_component_utils(checkers_test)

   constraint checkers_test_c {env_cfg.self_check_on == 1;}

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      set_type_override_by_type (async_par_bus_transaction#(`DATA_WIDTH)::get_type(),
                                 async_par_bus_err_trans#(`DATA_WIDTH)::get_type());
       set_inst_override_by_type ("*master_agent*", async_par_bus_master_driver#(`DATA_WIDTH)::get_type(),
                                                    async_par_bus_err_inj_mst_drv#(`DATA_WIDTH)::get_type());
       set_inst_override_by_type ("*slave_driver*",  async_par_bus_slave_driver#(`DATA_WIDTH)::get_type(),
                                                    async_par_bus_err_inj_slv_drv#(`DATA_WIDTH)::get_type());
       set_inst_override_by_type ("*passive_agent*", async_par_bus_monitor#(`DATA_WIDTH)::get_type(),
                                                    async_par_bus_err_check_mon#(`DATA_WIDTH)::get_type());
   endfunction
```

- Sets self-check bit of the interface to 1
- Creates all types of master & slave error trans. and sends them to sequencers

Compare in extended transaction

- Executed in SB for test checking the checkers

```
function bit async_par_bus_err_trans::do_compare(uvm_object rhs, uvm_comparer comparer);
   async_par_bus_err_trans trans_rhs;
   if(! $cast(trans_rhs, rhs))
      `uvm_fatal(get_name(), "Cast failed in transaction do_compare!");
    do_compare = 1;
   if(trans_rhs.error_type != NO_ERR) begin
      if(error_type != trans_rhs.error_type) begin
         `uvm_info(get_name(), $sformatf( "Error type mismatch: LHS = %s, RHS = %s",
                                  error_type.name, trans_rhs.error_type.name), UVM_LOW);
         do_compare = 0;
      end
   end
   else
      super.do_compare(rhs, comparer);
endfunction:do_compare
```

- Compares only error type field for error transactions
- Calls basic compare for legal transactions

Reset check

- Checks if all VIPs components react properly on reset
- Reset at the beginning of all the test and in the middle of transactions for reset test

```
// Reset
task async_par_bus_rst();
    `uvm_info(get_name(), $sformatf("Reset on time=%0d", $time ), UVM_LOW)
    async_par_bus_tb_top.MR = 0;
    env.scbd.reset();
    #(reset_duration*1ns);
    async_par_bus_tb_top.MR = 1;
endtask
```

```
// SB reset
function void async_par_bus_scbd::reset();
    `uvm_info(get_name(), "Scoreboard reset", UVM_LOW);
    ref_memory.delete();
    expected_tr.delete();
endfunction
```

- Driven directly from the test
- Reset test must empty SB list and re-initialize all its fields

Coverage

- VIP's deliverable coverage
  - Type of master transactions
  - Timing of master transactions
  - Type of slave transactions (for responders)
  - Timing of slave transactions (for responders)
  - Timing of ready/busy slave signal (for receivers)
- tEnv specific coverage
  - Type of injected protocol errors
  - Timing of injected protocol errors
  - Reset timing

Conclusions

- Advantages
  - Enables finding potential bugs in development phase
  - Improve quality of a VIP
  - Reduce debug time when VIP is used in verification environments
  - Can be used for complex protocols
  - Requires minimal code modification of deliverables (interface and monitor)
  - The same test environment and SB mechanism for all VIP's functions
- Disadvantages
  - Can't be applied to already written VIPs
  - Doesn't deal with features specific for some protocols
- Future work
  - Solution for UVM adapter checking
  - Checking VIPs for multi-layered protocols

Questions/Comments