2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Methodology for checking UVM VIPs

Milan Vlahovic, Veriest Solutions, Belgrade, Serbia (*milanv@veriests.com*)

Ilija Dimitrijevic, Veriest Solutions, Belgrade, Serbia (*ilijad@veriests.com*)

***Abstract*— Verification IPs (VIPs) are crucial components in verification environments. They play important role in verification of HW blocks/chips/SoC, but the issue is whether and how we verify them themselves? If we develop a VIP for some interface for specific project, we are going to work on, then we probably won't check it thoroughly – we will just integrate it in verification environment (VE) and debug together with the rest of the VE and design under test (DUT). If the goal is to create reusable component that will be used in many other projects, or we do it for a customer and the component will be our product, then we need to check all its functions. The paper presents one possible methodology for checking common functions that most of the VIPs written in UVM/SV have.**

Keywords—VIP, protocol, interface, checking, testing, transactions, monitor, driver, checkers, assertions, error injection, scoreboard.

I. INTRODUCTION

Every integrator and user of some verification IP (VIP) treats it as an already proven component which enables verification of some design under test (DUT) together with other components of verification environment (VE). When planning verification, verification teams almost never schedule time for VIP's debugging and fixing. Unfortunately, it often happens in practice that user is forced to debug the VIP as it doesn't behave as expected. That isn't easy task from many possible reasons:

- It can be tricky to debug code that someone else wrote.
- The VIP's files are encrypted.
- User isn't the VIP's protocol expert.

To avoid such situations, VIP developers must pay big attention on the checking of all main VIP's functions. Comprehensive VIP testing is necessary during its development phase.

II. WHAT TO CHECK?

Common functions of the VIPs that can be checked by this methodology are following:

- Driving transactions to interface's signals according to the protocol
- Monitoring interface signals and reporting transactions
- Checking protocol rules
- Reset

First 2 are the most important functions of VIPs in their usage in verification of some DUT. Driving is used to mimic HW block connected to DUT via the interface. Monitor is a passive and standalone component detecting transactions on the interface and sending them out for usage in VE components (global monitors, reference models, scoreboards, coverage…).

Protocol rules checkers enable finding possible bugs in DUTs that are in opposite side of the interface in VEs. If some protocol checker doesn't work correctly, we may miss DUT's bug, or monitor might be stuck or go to wrong state which makes debug harder. It is very important that VIP checks all the protocol rules and clearly reports if any of them is broken.

Forth function, not directly related to the interface's protocol, that we need to check is reset. As a global signal, reset is an input of our VIP's interface. VIP is required to properly react on applied reset – all its subcomponents must be taken to initial state and continue working after reset deactivation the same way as before.

III. VIP TEST ENVIRONMENT ARCHITECTURE

In parallel with a VIP development, self-checking testing environment (tEnv) needs to be created. Figure 2. shows the structure of the tEnv recommended by this methodology. Architecture of such tEnv enables easy and efficient checking of all above functions.

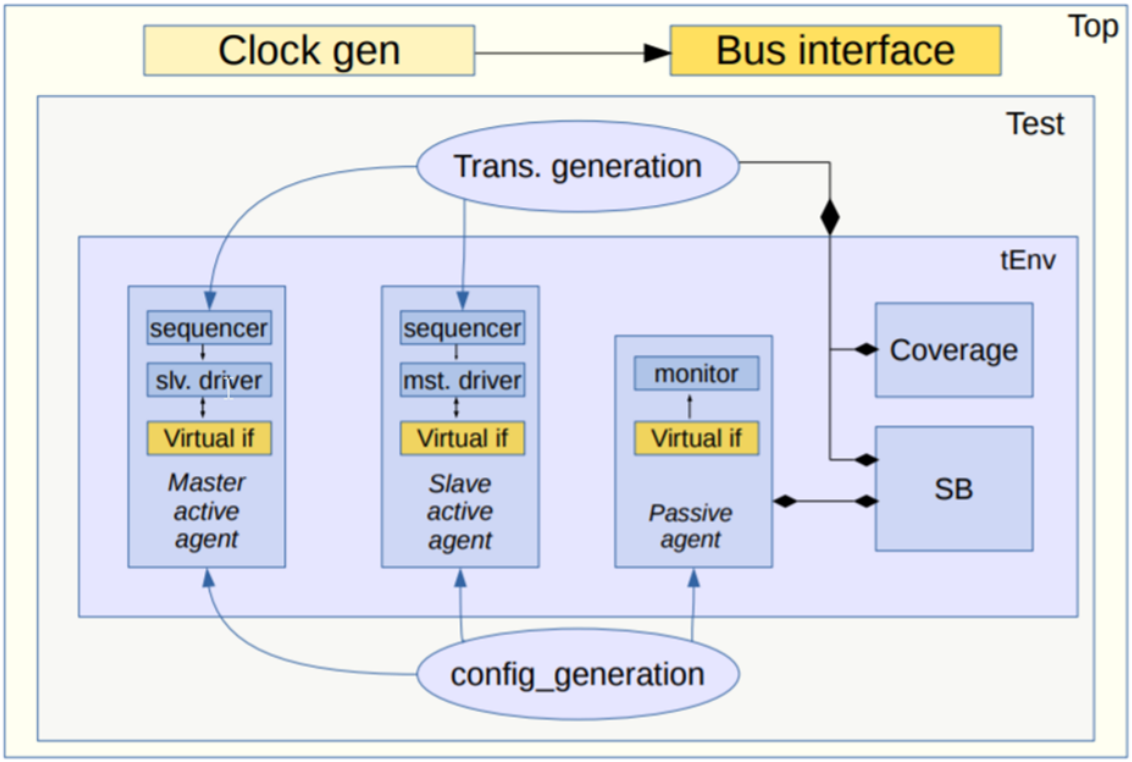

Figure 1. Structure of VIP testing environment

Most of the protocols specifies communication between 2 components usually called master and slave. According to this terminology the VIP's agent can be configured either as a slave or master depending on which side of the communication it is replacing and mimicking. In the tEnv there are active instances of both agents: master and slave. They are connected via the same SV interfaces (bus interface) instantiated in top SV module. Main usage of the agents is to check correctness of master and slave drivers. Monitor's checkers in both agents can be disabled as the monitor functions are checked in 3. agent which is configured as a passive component. It is done this way to prove that monitor is fully standalone component - capable of performing all its functions without any notifications neither from active components (driver and sequencer) nor sequences.

Scoreboard (SB) is an UVM component used to calculate expected monitor's transactions according to transactions generated by the test. When monitor reports a transaction, SB compares it against the expected one and reports an error in case of mismatch. It is connected to the test component via analysis port instantiated in tEnv and to the passive agent via analysis port of its monitor.

Config. generation represents tEnv configuration object (config_obj) creation and randomization with as same purpose as in VE for hardware verification. Configuration objects for all 3 VIP's agent are created in the tEnv's config_obj and passed to the agents from test by configuration database set. The config_obj has 2 additional fields: first one is assigned to bus interface's self-check bit, second is one is used to configure slave to work as a reactive responder.

Specific tests are created on the top of the tEnv for checking all VIP's functions. The tests don't call any sequence (neither from VIP's library nor specific tEnv sequence is being written), but they create transactions and send them directly to active agent's sequencers.

Coverage is an UVM component used to sample and collect coverage items that are not defined in VIP's coverage groups. It is connected to tEnv's analysis port such that all the transactions crated by test and written to the port enter the component.

IV. DRIVING AND MONITORING CHECK

First 2 functions are being checked together by the same tests. The tests create all possible legal interface transactions and send them directly to sequencers and SB (Figure 2). If driver doesn't work according to the interface protocol, monitor will report protocol error. Even the monitor doesn't detect the error, SB will report an error either by comparing the badly driven transaction from the test against first transaction well detected by monitor or by final check as at least one transaction will be left in SB. If drivers work as expected, but monitor detects transaction different from one sent by the test, then SB will report mismatch. If monitor doesn't detect any transaction or it is stuck, then we will have either SB mismatch upon first possible well driven and monitored transaction or by final check.

```
if(!std::randomize(address) with {address < `MEMSIZE;})
   `uvm_error(get_name(), "Randomization failed")

//WRITE request
if(!transaction.randomize() with {transaction.command == WRITE; transaction.address == local::address;})
   `uvm_error(get_name(), "Randomization failed");
send_master_trans(transaction);

//READ request
if(!transaction.randomize() with {transaction.command == READ_REQUEST; transaction.address == local::address;})
   `uvm_error(get_name(), "Randomization failed");
send_master_trans(transaction);
```

```
// Send master transaction to master sequencer and SB
task send_master_trans(async_par_bus_transaction#(`DATA_WIDTH) trans);
   an_port.write(transaction); // send transaction to SB
   master_seqr.execute_item(trans); // execute transaction on master sequencer
endtask
```

Figure 2. Creating and executing transactions from test

A. *Master commands check*

When checking master driver, slave agent must be configured such that it doesn't block master commands. If the slave is a receiver that generates ready/busy signal toward to master, slave should be configured to be always ready or busy with very small probability. If the slave is a responder, sending back responses with data to master, then slave must be configured to generate responses automatically (reactive responder). That means a reactive slave sequence (working in forever loop) is set as default sequence of slave sequencer.

B. *Slave transactions check*

If the slave is just receiver, then we need a test that creates ready/busy transactions and sends them to slave sequencer and SB. This way it is checked if the slave driver correctly drives ready/busy interface's signal and if the monitor detects it and generates and reports corresponding slave transaction.

If the slave is a responder, it usually works as a reactive responder. Reactive slave sequence is set as default sequence from the test if the corresponding configuration object field is set. If the responses are predictable (not randomized), then the SB is able to check them. One example is slave acting as a memory model. SB has an associative array with integer index as a referent memory. It stores in the array data when receiving write requests from the test. SB knows if the slave is configured to work as a reactive responder as it has a pointer to configuration object which is set from the test by configuration database. In case of reactive responder is configuredpon read

request SB calculates expected read response and pushes it to the expected list. That way SB checks if the reactive sequence generates responses and if the slave correctly stores data into its memory and returns them after read requests. SB's code implementing this is presented in Figure 3.

```
// Test port implementation
function async_par_bus_scbd::write_test(async_par_bus_transaction#(DATA_WIDTH) async_par_bus_trans);
    `uvm_info(get_name(), "Expected trans sent by test:", UVM_LOW);
    async_par_bus_trans.print();
    // Pushing trans. from test to expected list
    expected_tr.push_back(async_par_bus_trans);
    if(async_par_bus_trans.command == WRITE) // store write data into ref. memory
      ref_memory[async_par_bus_trans.address] = async_par_bus_trans.data_in;
    else if((async_par_bus_trans.command == READ_REQUEST) && config_obj.reactive_responder) begin
      // Expected reactive response calculation
      async_par_bus_transaction#(DATA_WIDTH) exp_rd_rsp_trans = new("exp_rd_rsp_trans");
      exp_rd_rsp_trans.command = READ_RESPONSE;
      exp_rd_rsp_trans.address = async_par_bus_trans.address;
      if(ref_memory.exists(async_par_bus_trans.address))
        exp_rd_rsp_trans.data_in = ref_memory[async_par_bus_trans.address];
      else begin
        `uvm_error(get_name(), $sformatf("Address %0d wasn't written to SB referent memory!",
                                         async_par_bus_trans.address));
      end
      // Pushing reactive response to expected list
      expected_tr.push_back(exp_rd_rsp_trans);
    end
endfunction
```

Figure 3. Expected transaction calculation in SB

A responder can generate various types of responses. Specific test controlling the responses instead of reactive sequence is needed to check if the slave driver is capable of driving all the types of responses and if the monitor correctly detects them. The test doesn't set default reactive sequence, but it generates any desired responses and sends them to the slave sequencer and to SB just after master transaction.

V. CHECKING PROTOCOL CHECKERS

This function is checked by intentional injection of protocol errors. Challenge is to make such tests pass without error messages reported by assertion or procedural checkers and to avoid simulation log search and waves debug when the checkers fire - correctly detect injected violations. Goal is to use already existing SB, that checks legal transactions, such that the SB error will be reported if some checker doesn't work well – doesn't fire when error is injected by the test. It is achieved by extending transaction, drivers and monitor and slight modification of the way assertion and procedural checkers are written.

A. Errors injection

VIP's transaction and drivers must support error injection. A single enumeration type containing values for all the errors checked by both, assertion and procedural checkers, is defined. New transaction class extending basic VIP's transaction, adds the field of new enumeration error type. Example of one extended transaction with new error enumeration type and error field is shown in Figure 4.

```
// 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;}
```

Figure 4. Error types in extended transaction

In the new transaction *do_compare()* function is overwritten such that it firstly checks the error type field of *rhs* operand. If it has non-zero value, it compares the error type fields. Otherwise, it calls *do_compare()* of basic transaction (Figure 5).

```
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
```

Figure 5. Compare implementation in extended transaction

Master and slave drivers must be extended by adding tasks dealing with all the types of error transactions. All the new classes, transaction with error types and drivers with task injecting errors, are defined in the tEnv's directory files and not a part of VIP's package, in order to avoid overhead in reusable and deliverable VIP's code. If the basic transaction and basic drivers already support injection of some errors, then the new classes add support just for remaining error types.

Task that implements driving of interface's signals in basic driver must be virtual. Extended driver, injecting errors, overrides the task such that it checks error type field from extended transaction. If it has non-zero value, it calls task injecting error, otherwise calls task from basic driver driving legal transaction (Figure 6).

```
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
```

Figure 6. Main driver's task override in extended driver

Generally, method of error injection depends on protocol and error type. As in many cases the error type is related to some legal transaction, task injecting errors can be minimized by reusing the code from basic driver as much as it is possible. Below example in Figure 7 shows how the unstable data during write command error is injected.

```
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
```

Figure 7. An expample of error injection

The task driving legal transaction from basic driver is called in parallel with error injecting code. While the basic driver is driving write command, error injecting branch is waiting for control signal activation and then changes the value on data bus after random time that is less than write command duration.

B. Procedural checkers

The way of checking protocol errors checkers written procedurally in monitor is very simple. At every point where the monitor detects some error, the error reporting by *`uvm_error* is replaced by calling new error processing function. The function has 2 input arguments: error message string and error number (procedural error ID). Enumeration value of every procedural checker error in added error type is equal to error ID of corresponding monitor checker. The checker modification and default implementation of new virtual function are shown in Figure 8.

```
@(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
```

Figure 8. Monitor's procedural checker modification

The error processing function is declared as a virtual function. In basic and reusable VIP monitor class it just prints *`uvm_error* message passed with function's input argument. In extended monitor the function doesn't print the error, but it creates transaction of error type corresponding to procedural error ID and writes the transaction to analysis port instead, that is described later in chapter D and shown in Figure 11.

C. *Assertion checkers*

Protocol checker can be written as SV assertion properties in VIP's interface. To enable monitor to check their correctness, an error flag of integer type and assertion self-check control bit are added in the interface. The error flag must have initial value 0, and self-check bit default value is 0. If an assertion fails, else branch implements desired behavior (Figure 9).

```
//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
```

Figure 9. Interface's assertion modification

As the else branch has the same format for all the assertion checkers, it is implemented as a macro which is called for every assertion with 2 arguments: error message string and error number (assertion error ID). Order of assertion checkers errors enumeration values in added error type matches the order of their error IDs.

If self-check bit isn't set, the error with given message is reported together with default message of the assertion. Otherwise, the default message is reported as a simple info and the interface error flag gets the assertion error ID value. Extended monitor detects the interface's error flag change and creates transaction of error type corresponding to the flag value (assertion error ID) and writes the transaction to analysis port which is shown in next chapter's Figures 10 and 11.

D. *Extended monitor*

Monitor class extending the base VIP's monitor is written to implement code that enables checking of all the protocol checkers. From the same reason as for extended transaction and drivers, the class is implemented in a separate file in tEnv directory and not a part of VIP's reusable package. The first function of the extended class is to observe interface's error flag. It is done in forever loop which is executed in parallel with run phase of the basic monitor (Figure 10).

```
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
```

Figure 10. Observing assertion errors in extended monitor

Once the interface flag is different from 0, the monitor calls the error processing function with error number argument equal to sum of assertion error ID and the last procedural error enumeration value from which the assertion errors enumerated in error type.

The second function of the extended monitor is to overwrite the error processing function which is declared as virtual in basic monitor. The new implementation of the function creates error transactions for both, procedural and assertion checkers, and sends them to analyses port (Figure 11) as it is already stated in above chapters B and C.

```
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
```

Figure 11. Creating error transactions and sending to SB in extended monitor

String *error_src* and printing the UVM info in above example are added just for debugging purpose.

E. Test checking the checkers

The test checking the checkers overrides the basic types of all transaction objects and master and slave drivers' instances with new types supporting error injection (Figure 12). Also, the type of monitor instance in passive agent is overridden with extended monitor class enabling checking of all protocol checkers. Self-check bit of the interface is set by constraining corresponding field in configuration object to 1.

```
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
```

Figure 12. Checkers test types overrides

In run phase the test multiple times creates and randomizes transactions of any error type and sends them to master or slave sequencer and to SB. The SB doesn't need any modification – it works the same way as for checking driving and monitoring legal transactions. If the basic VIP monitor does detect the injected protocol error, the transaction reported by extended monitor will match the transaction created by the test. If monitor detects transaction different from the error transaction sent by the test, then SB will report mismatch. If monitor doesn't detect any transaction or it is stuck, then we will have either SB mismatch upon first detected transaction by monitor or by final check.

VI. RESET CHECK

All the VIP components are obliged to react properly on reset – they need to break all ongoing actions and calculations, re-initialize all their fields and start from beginning after reset's de-activation.

Beside reset at the begging of all the tests, it is needed to have a reset in the middle of VIP's transactions. Reset test applies reset in the random time during master commands and slave responses. Reset actions in tEnv are presented in Figure 13. Reset signal is being driven directly from the test by accessing SV interface on the top TB. Test must delete all elements from expected list of the SB and re-initialize all its fields.

```
// 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
```

Figure 13. Driving reset from test and reseting SB

After some time, the reset is deactivated, and continues with master commands and slave responses. If some VIP's component isn't capable of processing reset by itself, SB will report mismatch during commands after reset or final SB check will fail.

VII. COVERAGE

As for HW verification, the most efficient way of VIP's checking is constrained random and coverage driven. Inputs that can randomized in above tests can be divided in following groups:

- 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)
- Type off injected protocol errors
- Timing of injected error
- Reset timing

Most of the groups should be part of reusable VIP's coverage which just needs to be enabled and collected. Only coverage related to injected errors (both type and timing) is specific for tEnv. It can be calculated in coverage component from extended error transactions generated by test. Coverage component is connected to the test via analysis port the same way as SB.

VIII. CONCLUSIONS AND FUTURE WORK

This methodology enables self-checking tEnv for all main and common functions of VIPs. By following the methodology, we can easily find many potential VIPs bugs and issues in development phase. This way, we improve quality of our product and reduce debug time when our VIPs are used in different levels of verification.

The methodology can be used for simple interfaces as well as for complex protocols. More complex interfaces would require bigger effort in tests creation, error injection and probably in SB's expected transaction calculation. General approach and tEnv structure stay the same as it is described in the paper.

Main advantage is that it uses, with minimal code modification in interface and monitor, the same tEnv structure and SB mechanism for checking all the functions. Once the tEnv and tests are complete, we can thoroughly check the VIP by running small regressions with VIP's and tEnV's coverages collected. As VIPs often contain parameterized classes, there might be bugs hidden for some parameter's values. Such bugs can be easily discovered by running more regressions in parallel, where every regression is run on compiled VIP's code for one legal set of parameters' values.

Disadvantage of the methodology is that checkers testing can't be applied on already written VIPs. Assumption is that VIP's development and checking are being done in parallel. VIP developers must know in advance how to write assertions and procedural checkers in order to enable their self-check in the tEnv.

Second disadvantage is that the methodology recommends solutions only for common VIPs functions. It doesn't deal with things specific for some protocol. Specific functions of some interfaces might require some modification in the methodology or in tEnv structure. For instance, I2C protocol specifies bus arbitration in multi-master mode. It is a feature of I2C VIP's sequencer and driver that can be checked with many master agent's instances in tEnv.

In the future, the methodology can be extended with UVM register adapter checking for the VIPs that offer it as a part of the product. Also, chalenge is to recommend solution for VIPs modeling multi-layered protocols.

REFERENCES

[1] Accelera, "SystemVerilog 3.1 Language Reference Manual", www.uvmworld.org

[2] Accellera, "UVM User's Guide, v1.2", www.uvmworld.org