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
- Enumeration value as an error ID
Extended driver injecting errors

- Main task in extended driver
- Error injection example
  - Using the code of basic driver as much as it is possible
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($time-time_check < 'WRITE_STROBE_MIN')
    process_error($format("WR command timing error: WE signal was active for %0d ns, but minimum is %0d ns", $time-time_check,"WRITE_STROBE_MIN"), 3);
  ```

- 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

```verilog
`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
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_ind_override_by_type("master_agent", async_par_bus_master_driver#(DATA_WIDTH):get_type(),
    async_par_bus_err_master_driver#(DATA_WIDTH):get_type());
  set_ind_override_by_type("slave_driver", async_par_bus_slave_driver#(DATA_WIDTH):get_type(),
    async_par_bus_err_slave_driver#(DATA_WIDTH):get_type());
  set_ind_override_by_type("passive_agent", async_par_bus_monitors#(DATA_WIDTH):get_type(),
    async_par_bus_err_check_moni#(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

```verilog
define 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)
   if (error_type != trans_rhs.error_type)
      `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
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
- Driven directly from the test
- Reset test must empty SB list and re-initialize all its fields

```verilog
// 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
```
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