2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA
Sept 25-26, 2014
Hotel Park Plaza
Bangalore, India

UVM usage for selective dynamic re-configuration of complex designs

Kunal Panchal, Applied Micro, Pune, India (*kunal.r.panchal@gmail.com*)

Pushkar Naik, Applied Micro, Pune, India (*pushkar.naik@gmail.com*)

***Abstract*—With the design industry moving towards software enabled hardware, demand for multi-subsystem designs has increased. The large number of configuration permutations used to switch subsystems in a single simulation requires the verification environment to mimic the same. With the evolution of such complex designs, the need for more robust and complete verification has become imperative. This obliged the verification industry to join hands and leverage the expertise across industry. Entrance of OOPS in verification methodology through System Verilog has addressed the reusability requirements. To add more, a variety of Methodology (like VMM, UVM, etc) are available that offer a standard approach to verification. These methodologies evolved around layering (phasing) mechanism as a basis of verification environment. Moreover, Verification Methodology has in-built basic components (like *uvm_object, uvm_component*, etc in UVM) which adhere to this layering mechanism. The present paper critically examines this supposition and certain other scenarios in real system that has its own verification challenges. One such scenario is dynamic re-configuration of complex designs where data can flow through multiple subsystems. The paper helps to achieve a deeper understanding of the mechanism used for verification of dynamic reconfiguration of SOCs.**

To verify a given chip with multiple subsystems, verification setup needs separate environments/agents for each subsystem to effectively verify diverse data paths within the chip and to leverage existing intellectual property. Each of these environments needs respective configuration classes. These configs ensure that all the components can be independently configured based on the testbench topology and the relevant DUT configuration that is currently being verified. Additionally, the testbench components need to be able to change the properties of the configuration classes dynamically based on the DUT responses and also be able to react to similar changes affected by the other TB components. Also, special care has to be taken during the reconfiguration process due to varied ordering requirements/cross-dependencies of each environment.

***Keywords*—dynamic reconfig, verification, UVM, complex SOC**

I. INTRODUCTION

In order to verify multi-subsystem chips, where data can change its course dynamically depending upon system requirements, dynamic verification platform is required. This platform should be robust enough to get/fetch data to/from selected environment(s). This leads to requirement of multiple environments in same test, each representing a subsystem.

Why dynamic reconfiguration is required in verification?

Referring to a networking application amongst many possible, network chips with humongous bandwidth and varied protocols are the need of the present time. Data centers/Service providers need a solution in which they can deploy nextGen system(s) alongside legacy system(s) to cater to both previous generation and next generation traffic. This drives the need to make chips with both new and old systems that are integrated in the form of subsystems, to be used depending on customer deployment. Figure 1 shows a chip with the dynamic reconfiguration requirement for two protocols handled by their respective subsystems.

Thanks to Mr. Pradeep Dharane, Director, APM Pune for sponsoring our travel/accommodation for DVCon presentation and encouraging us for same

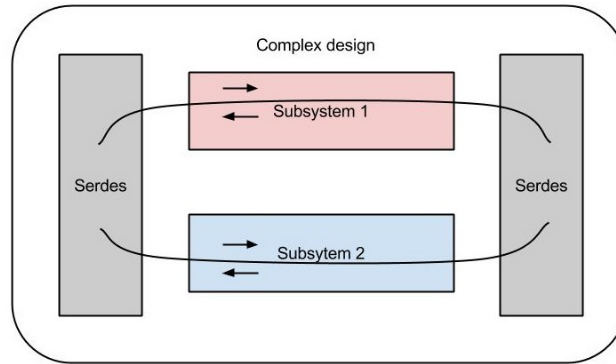

Figure 1 Multi-subsystem design

Chips with heavy complexity demand even more stable verification platforms.

With advent of software enabled hardware, chips have more and more subsystems integrated in same chip. By using proven verification environment for verification of such complex chips, we not only mitigate the risk of known DUT bugs, but also save a lot of time/effort for its development and rather focus on new sub-system(s) verification.

This paper portrays the basic scheme for achieving various test scenarios using the technique of environments switching. Paper also depicts how one should bind these environments with the DUT using virtual interfaces so that the dynamic change of traffic across all environments is permissible.

Tests typically have various mechanisms for communicating with the environments. Configuration can be set directly, modified via an API (application programming interface) or sent as a transaction through TLM port/export. UVM phasing mechanism dedicates a *connect_phase* to perform all port connection related operations. This approach works fine where no dynamic reconfiguration is necessary midway during a test scenario. But in case of complex designs currently under consideration, there is a requirement of performing new port connections during the *main_phase*. As UVM does not permit this, there was a need to device a workaround in order to achieve a dynamic port connection notion.

UVM primarily provides two configuration mechanisms – resource DB(mainly for static configs) and sequence item(mainly for dynamic configs). Generally static TB/DUT control parameters are part of resource DB. But current problem domain demands variation of few of these parameters during the simulation to handle dynamic reconfiguration scenario. A deeper look at an example system under verification will reveal the exact requirements that lead to the solution adopted.

II. DUT DESCRIPTION

Citing a networking application example as stated earlier, it is necessary to verify the dynamic rate reprovisioning of a DUT communication channel and its ability to adapt to the changes in protocol topology. Figure 2 depicts a network chip with 6 channels and a switching fabric to select respective subsystem(s) as per configuration.

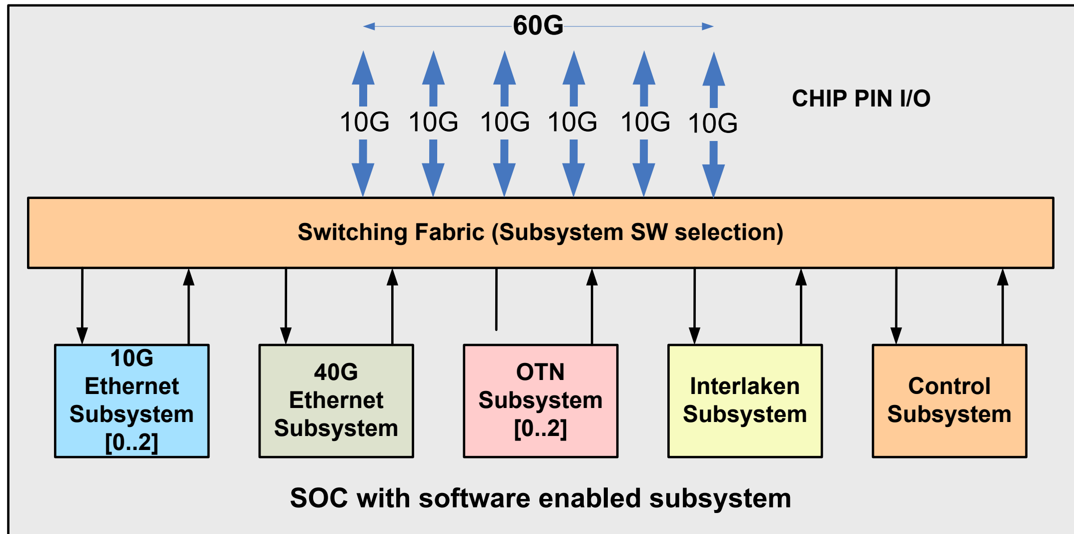

Figure 2 Network Chip supporting 60G BW (Various protocols can be used independently as per configuration)

Figure 3 shows how traffic flows through various subsystems on different channels. 10G Channels 0-3 undergo rate change to 40G. Channel 4 remains untouched before and after dynamic reconfiguration. Channel 5 depicts how protocol can be changed on same channel for same rate.

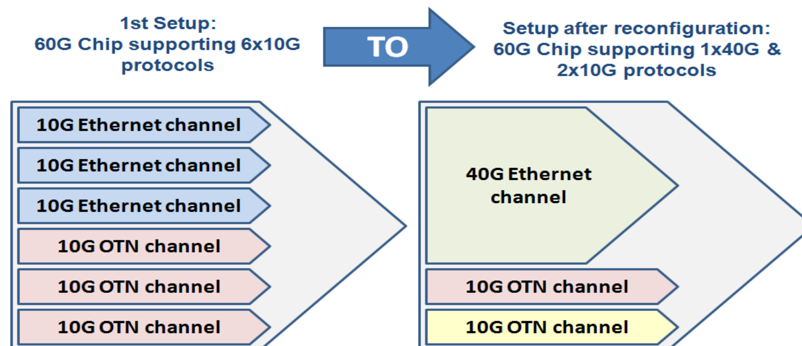

Figure 3 BW allocation in chip before and after dynamic reconfiguration. (Channel 4, 10G OTN is undisturbed)

III. Verification Considerations

Generally, multi- PHY protocols need separate subsystems in design for processing. Similarly, verification testbench can be architected to have each protocol handled by a separate environment. Based on a given datapath verification requirement at chip level, only the required subsystem environments like Ethernet, OTN and Interlaken will be instantiated. This is depicted in figure 4.

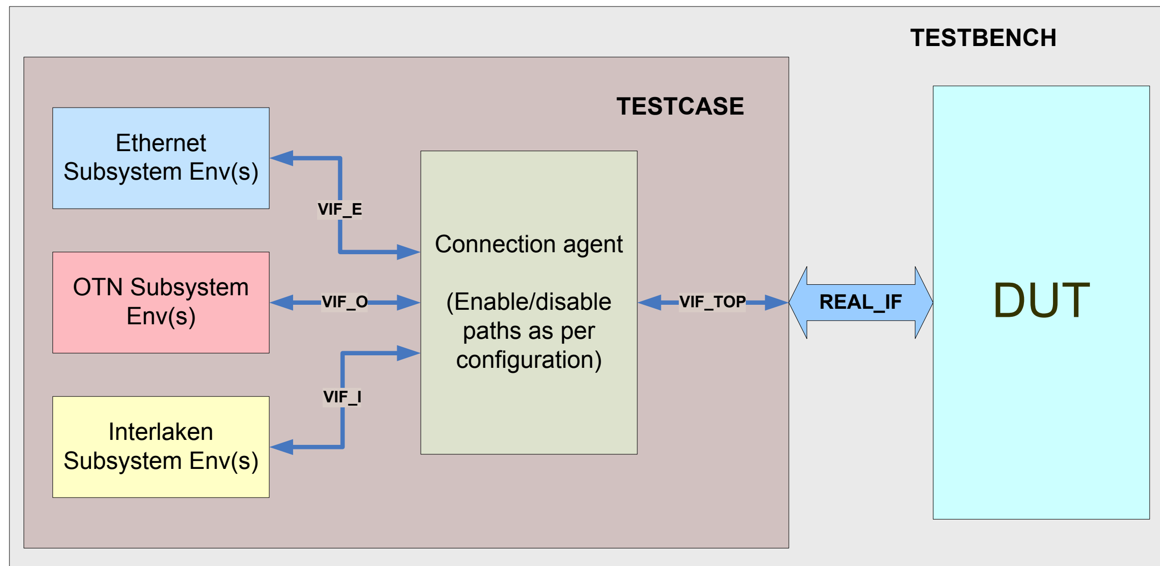

Figure 4 Multi-environment testbench

Each environment comprises of necessary UVM components like sequences, sequencers, scoreboard, driver, monitor, etc that is complete in itself to verify a DUT subsystem.

A. Basic recommendations

Following is the list of some basic recommendations to use for such a complex verification environment.

- As pointed earlier, a testcase can configure integrated environments in several ways. UVM suggests the usage of Config DB (*uvm_config_db*) to configure components.
- Several working scenario tests of active subsystem(s) could serve as a starting point to form a comprehensive dynamic reconfiguration scenario base test.
- Reset applied to various environments as well as DUT should be in sync to avoid any undesired state transitions of a given component with respect to others. Thus all these resets should be placed in UVM *reset_ phase.*
- During dynamic reconfiguration, DUT might pose certain requirements of holding one or more blocks under reset. However, since, the *reset_phase* has already passed, these particular reset requirements have to be handled in the *main_phase* itself, carefully.
- Clocks supplied to each environment for driving various interfaces should be configurable with respect to parameters like frequency, duty cycle, enable/disable etc, since after reconfiguration, the DUT might have new requirements on these.
- An SV interface with all DUT I/O signals is created. A top level virtual interface instance is created of this interface type, which finally connects to DUT’s real interface. Each environment has its own local virtual interface instance created. It is recommended that these instances are of same type for convenience of its connection to top level VIF instance, directly. Else, different type of environment level virtual interfaces need to explicitly map its signals to corresponding signals in top level virtual interface. If a common interface type is chosen, as recommended, the respective environments will access only the relevant signals in that interface, that they are suppose to.
- In a dynamic reconfiguration scenario the data paths before and after could be different. This may lead to new scoreboards becoming active and the old ones getting squelched. Thus one may need a control over enabling/disabling or squelching/flushing of scoreboards on need basis.
- The lifetime of sequences associated with an environment being phased out has to be so configured such that the sequence(s) end(s) correctly at the time of reconfiguration for a graceful exit. Also the starting of

the sequences associated with the newly activated environment has to be handled carefully such that any DUT/TB configurations are done prior to this.

- Any common configuration tasks called in *configure_phase* might be required to be called again in *main_phase* for the newly activated environment. It is better to place these common tasks in base test for all derived tests to leverage upon.

B. Limitations Faced

Re-iterating the problem statement - the puzzle to be unfolded is that in the dynamic re-configuration of multi-subsystem scenario, the verification setup would change its mode of operation based on the configuration provided runtime.

Listed below are some of the challenges likely to be faced with conventional UVM approach while trying to address the above problem.

- As this reconfiguration involved multi-subsystem environment change runtime, the overall environment variation required to be handled was quite sizeable. The above variation involves various environment components to be re-configured during simulation, thus complicating the configuration requirements of these otherwise standard components.
- As known already, System Verilog does not impose any limitation on initialization of any instance at any point of time during simulation, while UVM methodology, on the other hand, does have this limitation. It doesn't allow creation of any components outside *build_phase*, hence creation of components at the time of reconfiguration makes it imperative to look for a work-around. For example, while re-configuring DUT in *main_phase*, new environment(s) cannot be created to replace the existing environment(s) that were created during *build_phase*.
- System Verilog mail box connections can make/break at any time to support our purpose of dynamic reconfiguration. While UVM puts a restriction on this and prohibits any change in TLM port connections post *connect_phase.*
- UVM's phase looping feature technically allows jumping back from *main_phase* to *build_phase.* However, in the current networking example, during dynamic reconfiguration, it is necessary to ensure that any channel that is not a part of reconfiguration process, should not loose data integrity at any point of time during testing. This demands that its respective data sequence(s) continue to run in the main phase itself, while for channels being reconfigured, we need to jump to *build_phase*. Thus phase looping cannot be used as partial jumping is not supported.

IV. Solution Proposed

Solution to this problem resembles the RTL design approach using switching of subsystems. All required environment(s) in the entire test scenario are created during *build_phase* itself. Also, TLM ports, if any, should be connected during the *connect_phase* itself, for all the environments. The above two requirements come from UVM restrictions stated earlier that are overcome by playing dynamically with the virtual interface(s) instead. As described in figure 4, connect each environment's local virtual interface to a top level virtual interface (vif_top) in *main_phase*. Also as traditionally done, all real interface connections to their top level virtual interface counterparts will be statically taken care of at testbench level without any relation to UVM phasing. Flow chart in figure 5 explains this crux of solution.

A. Flow Chart

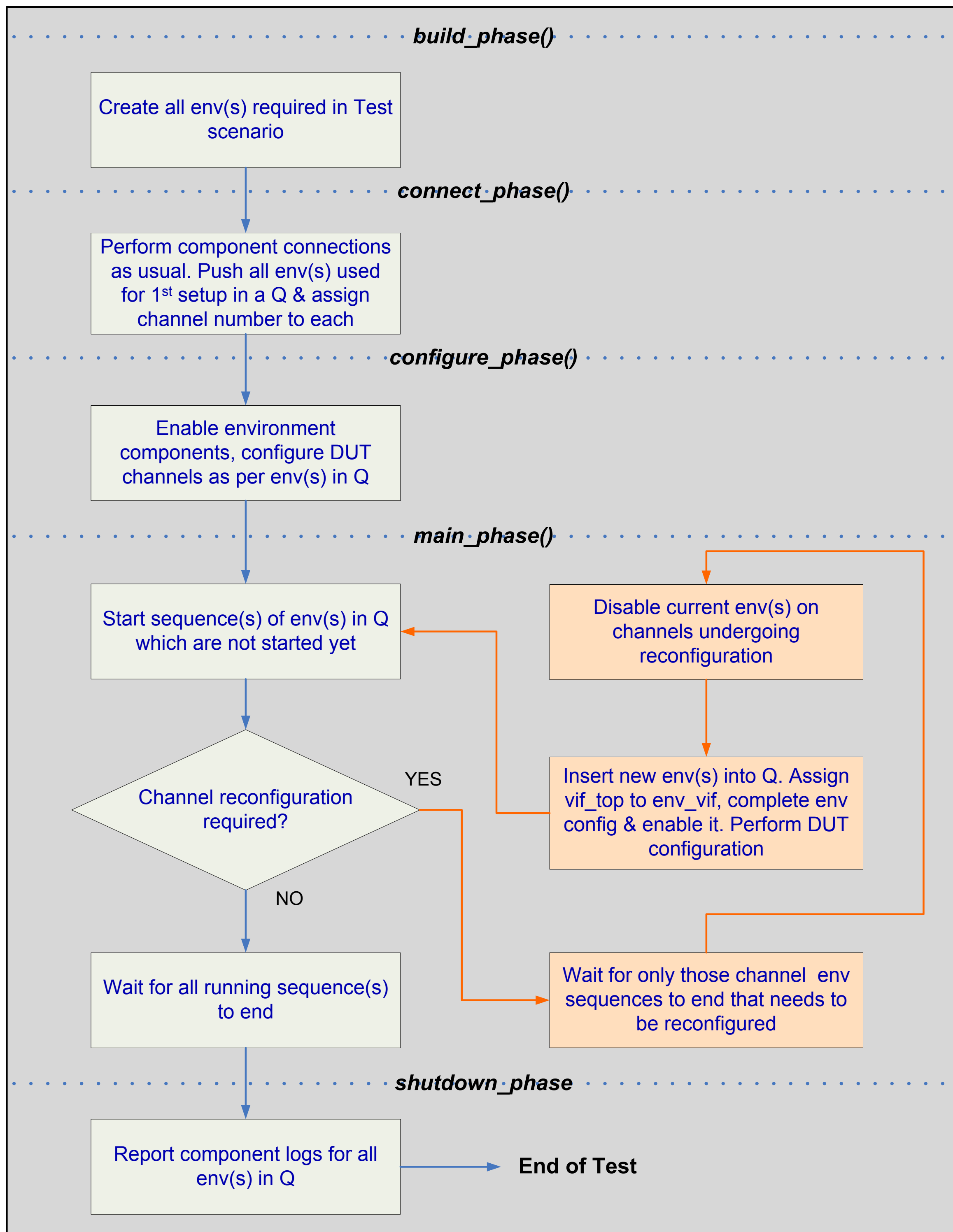

Figure 5 Flow chart of a single channel re-configuration

B. Sample Code

First of all, we need to identify the number of environments required to be integrated in the test. They should be properly initialized and their checkers should be connected with their respective agents. Sample code for *base_test* and derived test is being shown below mainly for their build, connect, configure and main phases.

Note that, only the relevant sections of code are shown below, while the generic part is shown as comments due to size constraints.

Base test code to determine environment(s) required to be created/connected as per derived test scenario. Methods specific to dynamic reconfiguration requirement are also placed here.

```
//Abstract class to ensure that it is not directly instantiated, but always extended to create a specific test
//scenario. Respective clocks and resets are passed to each environment created
virtual class base_test extends uvm_test;
  `uvm_component_utils(base_test)
  //SV queue of base env type for storing derived environment handles during simulation
  base_env env_channel[$];
  //Global variable to keep count of total environments
  int m_channels;
  //Top virtual interface used for connection with DUT interface
  top_interface vif_top;
  …
  //Following task has to be virtual so that new environments can be added
  //dynamically during simulation in the queue
  //This task finds already created environments by derived test, and inserts them into a queue
  //that will be used for enabling/disabling the channel environment(s)
  virtual function void find_env_channels();
    base_env env;
    string name;
    m_channels = 0; //Count for total channels

    //Use UVM method to determine the names of all objects created by base_test and its derived classes
   if (get_first_child(name)) begin
     do begin
       //Make sure handle returned is a child of base_env type
       if ($cast(env, get_child(name))) begin
         env_channel.insert(m_channels++, env);
       end
     end while (get_next_child(name));
    end
    uvm_report_info("base_test", $psprintf("Found %1d env channels", m_channels));
  endfunction

  //to connect clocks, tb config, dut reg config, interrupt interface etc
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    //perform any required tlm port connections
    …
    …
    find_env_channels();
  endfunction

   //This will enable/disable components of environment
  function void enable_env(base_env env);
    //This will enable clock generation based on is_active or any other designated config bit
    env.enable_clocks();
    //This will enable driving logic by checking is_active or any other designated config bit
     env.enable_driver();
    …
    …
  endfunction

  function void disable_env();
     //This will disable clock generation based on is_active or any other designated config bit
    env.disable_clocks();
```

```
  //This will disable driving logic by checking is_active or any other designated config bit
  env.disable_driver();
  …
  …
endfunction

 //Environment configure task using channel env queue
 task configure_phase(uvm_phase phase);
  for(int i=0; i<m_channels; i++)
    fork automatic int j = i; begin
      enable_env(env_channel[j]);
      //Set configuration of env components like stimulus paths, monitoring paths, scoreboards, etc
     env_channel[j].configure_phase(phase);
      m_sem.put();
     end
    join_none
  //Wait for all new reconfiguration channels to finish above task
  m_sem.get(m_channels);
endtask

  function void config_ch_env(base_env env_channel, base_cfg env_cfg, ctrl_vif dut_reg_cfg);
    //dut_cfg is handle of control interface of DUT e.g. ahb, spi, i2c, pcie, etc
   env_channel.dut_cfg      = dut_reg_cfg;
   //env_cfg is handle of configuration class
   env_channel.env_cfg      = env_cfg;
   …
   …
   //following is env's virtual interface connection with top level virtual interface
   //note that this virtual interface is potentially capable of carrying info of all channels
   //but environment will choose to use info of only those channels for which it is configured
  env_channel.env_vif = vif_top;

endfunction

//This method will take care of starting traffic on new channels which will be called from derived test
//Here re_channel is channel number on which new env will re-start. It will be called in the derived test
function void start_traffic(int re_channel, base_env ch_env, base_cfg cfg, ctrl_vif reg_cfg);
  $display("Starting test on channel =%0d", re_channel);
  //Inserts env of channel(s) dynamically into the queue as per reconfiguration requirement
  //This will set env to drive on "re_channel" only out of all channels present in virtual interface
  //e.g. during reconfiguration of 10G channels 0-3 to 40G channel 0, environment is configured
  //with "ch_env.channel=0", though the data will actually be driven on channels 0-3
  ch_env.channel = re_channel;
  env_channel.insert(m_channels++, ch_env);
 //This will assign virtual interface with top interface for data communication
  config_ch_env(ch_env, cfg, reg_cfg);

endfunction

//Apply reset, configure and enable traffic flow for the new channel. It'll be called in the derived test
task run_traffic(uvm_env ch_env);
  //Reset only the state machines; so already configured dut register values should not be lost
  assert_soft_reset(ch_env.channel);
  //Enable env to be used
  enable_env(ch_env);
 //Configure clock period, duty cycle, etc
  ch_env.setup();
  //Provisioning the dut subsystem
  ch_env.configure_dut();
  //Deasserting soft reset
```

```
    //Note that as per DUT's soft reset requirement the procedure given here might vary
    deassert_softreset(ch_env.channel);
    //Starting the test sequence(s)
    ch_env.start_sequence();
  endtask

endclass
```

The switching technique referred in *base_test* will also be able to start/stop clocks to an environment. Example derived test below shows how 10G Ethernet channels 0-2 & 10G OTN channel 3 is replaced by a single 40G Ethernet channel(consuming four 10G channels) and 10G OTN channel 5 by 10G Interlaken channel.

```
//Derived test for switching of env(s). This code gives an idea of how a running
//environment on one channel is replaced by another environment, dynamically.
// Similar approach can be used for verification of multi-subsystem chips.
class dynamic_test extends base_test;
  `uvm_component_utils(dynamic_test)
   //instantiate required environment(s)

  // 1st setup needs three 10G ethernet channels and three 10G OTN channels
  //Note that these channel env's are derived from same base_env
  enet_env enet_channel[3];
  otn_env otn_channel[3];
  // 2nd setup needs one 40G ethernet channel and one 10G interlaken channel
 // after dynamic reconfiguration
// Note that one 10G OTN channel from 1st setup remains undisturbed throughout
  ikn_env ikn_channel;
  enet_40g_env enet_40g_channel;

  //Arrays of config classes for each environment
  enet_cfg enet_config[3];
  otn_cfg otn_confgi[3];
  ikn_cfg ikn_config;
  enet_cfg enet_40g_config;

  //Arrays of ctrl cfg for each environment derived from ctrl_vif
  apb_vif apb_cfg[4];
  spi_vif spi_cfg[4];

  //Variable depicting which channels undergo reconfiguration
  bit reconfig_map[6] = {1,1,1,1,0,1};
  //Variable to tell total number of channels affected during reconfiguration
  int reconfig_channels = 5;

  //In build_phase create all channel envs
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    //create all 6+2 environments like
    enet_channel[0] = enet_env::type_id::create("enet_channel_0", this);
    enet_channel[1] =enet_env::type_id::create("enet_channel_1", this);
    enet_channel[2] = enet_env::type_id::create("enet_channel_2", this);
    otn_channel[0] = otn_env::type_id::create("otn_channel_0", this);
    otn_channel[1] = otn_env::type_id::create("otn_channel_1", this);
    otn_channel[2] = otn_env::type_id::create("otn_channel_2", this);
    enet_40g_channel = enet_40g_env::type_id::create("enet_40g_new_channel_0", this);
    ilkn_channel = ikn_env::type_id::create("ilkn_new_channel_5", this);
    …

  endfunction
```

```
//Assign parameters like interface and config to channels to be reconfigured
//Overriding the base_test method
function void find_env_channels();
//Add only those channel envs which are configured at start of simulation
//channel envs to be reconfigured later should not be inserted at this time
  start_traffic(0, enet_channel[0], enet_config[0], apb_cfg[0]);
  start_traffic(1, enet_channel[1], enet_config[1], apb_cfg[1]);
  start_traffic(2, enet_channel[2], enet_config[2], spi_cfg[0]);
  start_traffic(3, otn_channel[0], otn_config, spi_cfg[1]);
  start_traffic(4, otn_channel[1], otn_config, spi_cfg[2]);
  start_traffic(5, otn_channel[2], otn_config, spi_cfg[3]);

  uvm_report_info("dynamic_test", $psprintf("Found %1d channels", m_channels));
endfunction

task main_phase(uvm_phase phase);
  //Semaphore to wait for all active envs to complete, post reconfiguration, to decide upon test exit
  semaphore exit_test = new();
 //Semaphore to wait for an env sequence completion for reconfiguration
 semaphore m_sem = new();
  int my_channels = m_channels;
  for(int i=0; i<m_channels; i++) begin
    $display("Starting of ch %d channel_env", i);
    //Start the channel env(s) for 1st setup
    fork begin
        env_channel[i].start_sequence();
        //Once channel env's sequence ends, check if reconfig required on that channel(s)
        if(reconfig_map[i]) m_sem.put();
        else exit_test.put();
      …
     end
    join_none
  end
  //Wait for channel env to complete sequence where reconfiguration needs to be done
  m_sem.get(reconfig_channels));
  re_test();
  //Wait for all the untouched channel env(s) to end
  exit_test.get(my_channels - reconfig_channels);
endtask

virtual task re_test();
  //Save the 1st setup's active channel count
  int existing_channel = m_channels;
  //Disable existing channel env components which are reconfigured 0,1,2,3,5
  disable_env(env_channel[0]);
  disable_env(env_channel[1]);
  disable_env(env_channel[2]);
  disable_env(env_channel[3]);
  disable_env(env_channel[5]);

  //Enable channel env(s) for reconfiguration on channel 0 and 5
  //Note that 40G channel will occupy four 10G channels
  start_traffic(0, enet_40g_channel, enet_40g_config, apb_cfg[2]);
  start_traffic(5, ikn_channel, ikn_config, apb_cfg[3]);
  //Run the loop for the difference of 1st setup and 2nd setup channel env(s)
  for(int i=existing_channel; i<m_channels; i++ )
    fork begin
      int automatic new_channel = i;
      run_traffic(env_channel[new_channel]);
    end
```

```
    join_none
  end
  endtask
endclass
```

env_channel queue used above to save channel environment handles, has to maintain the handles of both first and second setup together at the time of *shutdown_phase/report_phase*. This is to ensure that Scoreboarding of both the setups happen correctly as a test conclusion.

Above sample code portrays a single use-case of dynamic reconfiguration. Nonetheless, there may be extensive combinations of rate and protocol variations possible in dynamic reconfiguration. Manual creation of all these scenarios is laborious. Sample code can be extended to use randomization feature of SV so that aforesaid manual efforts can be avoided. This is explained below.

```
//Pseudo code for randomization of initial/reconfig channels
//Use MAX_CHANNELS defined for a given chip
//LOW_BW defines one unit of channel BW; LOW_BW = 1
//HIGH_BW defines number of LOW_BW units required to make one unit of HIGH_BW
//In our case HIGH_BW is defined for 40G; HIGH_BW = 4;

task  randomize_channel_env();
  //code to determine total number of active channels in current setup out of MAX_CHANNELS
  //Inactive channels (MAX_CHANNELS – active channels) will not carry any traffic
  randomize(active_channels);
  //unintended channel represents an inactive channel before reconfiguration and untouched
  //channel after reconfiguration
  unintended_channels = MAX_CHANNELS – active_channels;
  //Loop for number of unintended channels
  while(unintended_channels--)  begin
    //p refers to randomized position of unintended channel within total number of channels
    get_unintended_position(p);
    if( ((p-HIGH_BW) >=0) OR ((p+HIGH_BW) <= MAX_CHANNELS) )
    add_HIGH_BW_to_rate_constraint();
    else rate_contraint = LOW_BW; //Fixed to this single value
  end

  //return the rate/protocol values for all channels, but only relevant channel values as per rate will be used
  randomize(rate, protocol);
  for(int channel=0; channel<MAX_CHANNELS;)begin
    //this will return created env handle based on rate and protocol passed in argument
    //env_Q will store all the handles returned that are declared in base_test
    env_Q.push_back(get_env_handle(rate[channel], protocol[channel]));
    if (rate[channel] == HIGH_BW) channel +=HIGH_BW;
    else channel += LOW_BW;
  end
endtask

  //Following code should be placed in build phase of derived test
function void build_phase(uvm_phase phase);
  bit reconfiguration_required=1;
  super.build_phase(phase);
  do begin
    randomize_channel env();
    randomize(reconfiguration_required);
  end while(reconfiguration_required);
  …
endfunction
```

Note that the given example *base_test* class can also be used for verification of an individual subsystem in chip without any reconfiguration scenario.

V. CONCLUSION

In multi-subsystem chip scenarios, it is necessary to verify the dynamic reprovisioning of components and the ability of components to adapt to changes in the protocol topology. Circumventing the UVM reconfiguration reservations, the paper describes how a verification setup is still able to verify the complexity of dynamic reconfiguration in a multi-subsystem chip. This mechanism of encapsulating dynamic control of environments in a test scenario will find applications in various verification setups where the data flow is dependent on configuration information.

ACKNOWLEDGMENT

The authors would like to acknowledge the help extended in understanding the base testbench architecture by APM team member, David Cornfield, which provided us our starting platform. We would also like to thank our APM team members for helping us understand any subsystem specifics required to be dealt with, for reconfiguration. Special thanks to our Team members Nisha Mallya & Shailesh Wagh for their valuable review inputs. We would also like to thank online UVM community for responses posted that helped us. And finally an all-time thanks to Accellera for their continued effort for betterment of verification methodologies.

REFERENCES

[1] Accellera, “Universal Verification Methodology (UVM) 1.1 User’s Guide, May 2011

[2] Universal Verification Methodology (UVM) 1.1 Class Reference, updated September 19, 2012

[3] IEEE Std 1800-2009 "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language"http://dx.doi.org/10.1109/IEEESTD.2009.5354441