AMIQ_ECTB – Release Alpha 1.0

Conference presence

This article is a follow-up on the paper presented at DVCon EU 2022 entitled “How creativity kills reuse – A modern take on UVM/SV TB architectures”. As promised at the presentation, albeit with some delay, we have uploaded our implementation together with a usage example. This article covers a small introduction, explanations on the contents of the library as well as a small user guide to get you started.

Introduction

We’re pleased to announce that we released the first version of our UVM/SV TB architecture, featuring a framework that enables an externally control layer on top of a UVM TB. This library is the result of years of work on corporate projects that touch the boundaries of what is possible with the current available tools.

A simple search online online for something in C or Python is probably going to result in an open-source implementation to nearly any popular problem. Unfortunate that’s not the case with HDLs and aggregate languages or tools.

“Amiq externally controlled TB” (amiq_ectb) came to life from the idea of enabling ourselves to create our own tools. We wanted to be able to use existing technics and algorithms available in other languages without the use of DPI and multi-language support and without having to undergo massive changes in how the TB is structured.

To conclude this introduction, we wanted to expose the verification process to processes outside the simulator world. We wanted a higher control layer that can be easily organized and automated. We wanted exactly what PSS promised, but failed to deliver. And we learned something from that: user experience and the ease of adoption is far more important than the list of features and the flashy graphs.

User Guide

Structure

The structure of the available library features the following dir’s:

- scripts – Features different scripts that we are using to automate the creation of scenarios as well as coverage closure (This will be the subject of future articles)
- sim – Contains some examples of sim_args and a regression file for a given simulator

- sv – Source directory of the amiq_ectb library. This is the dir that contains the packet to be imported for the library usage. Compile and import the amiq_ectb_pkg.sv in your TB.
- tb – An example TB created to test and showcase the usage of amiq_ectb framework.
- tb/src – Sub-dir of the example TB that contains the TB top.
- tb/sv – Sub-dir of the example TB that contains the environment and other components excluding the VIPs
- tb/sv/seq_lib – Sub-dir of the example TB that contains the virtual sequences that instantiate, constraint and use the VIP "physical" sequences.
- tb/sv/tb_vip_blue – Sub-dir of the example TB that contains a generic VIP.
- tb/sv/tb_vip_purple – Sub-dir of the example TB that contains a generic VIP.
- tb/sv/tb_vip_red – Sub-dir of the example TB that contains a generic VIP.
- tb/tc – Sub-dir of the example TB that contains the test and an example collection of plusargs that build an env and constraints and schedules the sequences to create a scenario.

Available classes

The library features base classes that would replace the basic uvm_* components that you would normally extend from. These classes are the following:

- amiq_ectb_component – Extended for any misc component like a custom monitor, a scoreboard, a coverage collector, a responder or any other components that might have nested objects/components and variables that require control from an upper layer. Features a predefined build_phase that enable the creation of components dynamically with our framework.
- amiq_ectb_environment – Similar to the component, features the same functions and has the same capacity, but is built on top of uvm_env instead of uvm_component. The main class that has to be extended for creating a completely dynamic and scaling environment.
- amiq_ectb_object – Base class for any object. This offers the capacity to control based on the scenario the value of any variable that has been registered in the flow.
- amiq_ectb_sequence – Similar to the object as far as functionality goes, but usable for sequences.
- amiq_ectb_test – The most important piece of the puzzle, and the base class for the test. Extending from this object allows our sequence scheduling system to be used and allows the dynamic creation of any scenario possible. In this architecture, this is the base test and all scenarios are being dynamically defined from text files or command line.

How to get started

As the framework is built on top of uvm, the first and main requirement of adopting this work flow is to extend the relevant classes from the amiq_ectb classes, instead of the uvm ones. These classes

provide helpful functions which aim at delivering a dynamic TB and an external control layer for the environment and scenarios.

Dynamic environment

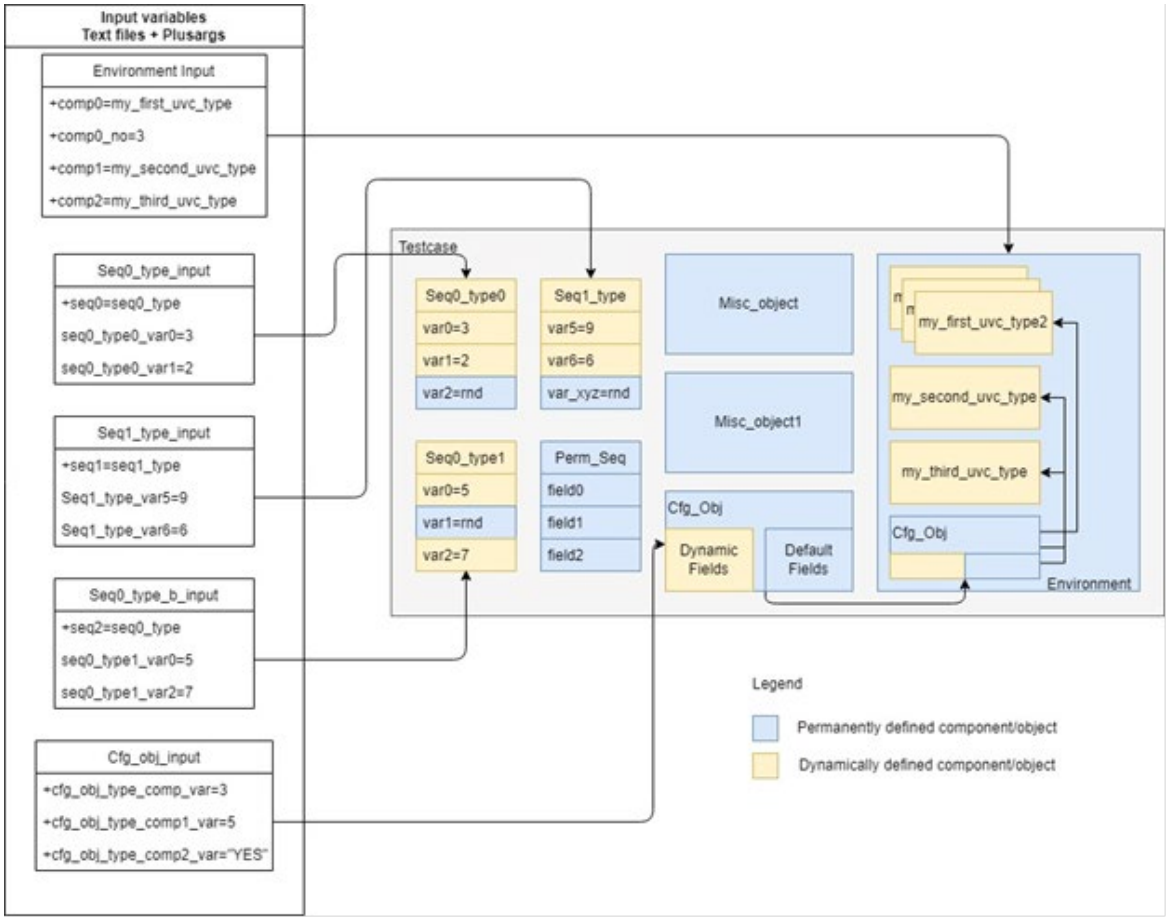

Figure 1. Dynamic Environment Example

The first important feature is the ability to create completely dynamic environments that can scale up and down based on a "per scenario" basis, as required. The above figure pictures an environment that is formed of both permanent and dynamic components.

Permanent components are the ones that are always present and are instantiated in a hardcoded manner. Dynamic components on the other hand, are scalable components that can be instantiated or not based on the scenario that's being run. This chapter will go in-depth into how all of this is possible.

```
class amiq_dvcon_tb_env extends amiq_ectb_environment;
```

Figure 2. Environment extended from amiq_ectb_environment

To create a dynamic environment, the first step is to extend the environment from the "amiq_ectb_environment" class (Figure 2).

See "../amiq_ectb/tb/sv/amiq_dvcon_tb_env.svh" for a usage example.

```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    pre_create_objects();
    push_all_objs();
    create_objects();
    post_create_objects();


    pre_create_components();
    push_all_comps();
    create_components();
    post_create_components();

endfunction : build_phase

virtual function void pre_create_objects();
endfunction

virtual function void post_create_objects();
endfunction

virtual function void pre_create_components();
endfunction

virtual function void post_create_components();
endfunction
```

Figure 3. Predefined build_phase in amiq_ectb_environment

This base class contains functions for dynamically creating components based on external input (plusargs) and a base definition for the build phase that call all the relevant functions in the right order. (Figure 3)

Apart from calling relevant functions, the build phase also contains empty hook functions for proprietary user code. This are called before and after the creation phase for objects and components.

The objects are created first, as the process can be used to create configuration objects for VIPs. The post_create_objects() function can be used to set the create objects to the config db as needed.

For in-depth definitons of the push/create objects/components have a look at files:
"../amiq_ectb/sv/amiq_ectb_obj_create_functions.svh", and
"../amiq_ectb/sv/amiq_ectb_comp_create_functions.svh"

```
virtual function void post_create_components();
    super.post_create_components();
    cast_agents();
    configure_agents();

    // We cannot have a working TB without a sequencer, so we create it outside
    // of the plusarg dynamic scheme
    virtual_sequencer = amiq_dvcon_tb_sqr::type_id::create("virtual_sequencer", this);
endfunction : post_create_components


function void cast_agents();
    foreach(components[i]) begin
        case(components[i].get_type_name())
            "amiq_dvcon_tb_vip_red_agent": begin
                amiq_dvcon_tb_vip_red_agent proxy_agent;
                $cast(proxy_agent, components[i]);
                my_red_agents.push_back(proxy_agent);
            end
            "amiq_dvcon_tb_vip_blue_agent": begin
                amiq_dvcon_tb_vip_blue_agent proxy_agent;
                $cast(proxy_agent, components[i]);
                my_blue_agents.push_back(proxy_agent);
            end
            "amiq_dvcon_tb_vip_purple_agent": begin
                amiq_dvcon_tb_vip_purple_agent proxy_agent;
                $cast(proxy_agent, components[i]);
                my_purple_agents.push_back(proxy_agent);
            end
            // Keep in mind, that components that have to always exist, can be created separate
            // This serves as an example of an env that can be created without a coverage collector
            "amiq_dvcon_tb_coverage_collector": begin
                $cast(cov_collector, components[i]);
            end
        endcase
    end
endfunction
```

Figure 4. Hook function used for casting dynamic components to relevant type instances

Once this is understood, the next step is to prepare the environment to work with a changing number of objects/components. As seen in the “amiq_dvcon_tb_env.svh” example, because the components/objects are dynamic, we need to create smart functions that parse the object/component queue and cast the instances to their relevant types. (Figure 4)

Because the names are an important part of utilizing the factory, the same names can be used to correlate certain types of objects to the base types.

Also, this is not a mechanism that you should feel constrained by. There are no requirements for this type of mechanic to be present every time a component of object is created. It is very much possible to have a hybrid environment that has static defined objects/components as well as dynamic ones.

```
function void configure_agents();
    if(my_red_agents.size()>1)
        foreach(my_red_agents[i]) begin
            amiq_dvcon_tb_vip_red_cfg_obj proxy_red_agent_cfg;
            proxy_red_agent_cfg = red_cfg(i);
            uvm_config_db#(amiq_dvcon_tb_vip_red_cfg_obj)::set(this, $sformatf("*red_agen
        end
    else if(my_red_agents.size()==1) begin
        amiq_dvcon_tb_vip_red_cfg_obj proxy_red_agent_cfg;
        proxy_red_agent_cfg = red_cfg(0);
        uvm_config_db#(amiq_dvcon_tb_vip_red_cfg_obj)::set(this, $sformatf("*red_agent*")
    end
    if(my_blue_agents.size()>1)
        foreach(my_blue_agents[i]) begin
            amiq_dvcon_tb_vip_blue_cfg_obj proxy_blue_agent_cfg;
            proxy_blue_agent_cfg = blue_cfg(i);
            uvm_config_db#(amiq_dvcon_tb_vip_blue_cfg_obj)::set(this, $sformatf("*blue_ag
        end
    else if(my_blue_agents.size()==1) begin
        amiq_dvcon_tb_vip_blue_cfg_obj proxy_blue_agent_cfg;
        proxy_blue_agent_cfg = blue_cfg(0);
        uvm_config_db#(amiq_dvcon_tb_vip_blue_cfg_obj)::set(this, $sformatf("*blue_agent*
    end
    if(my_purple_agents.size()>1)
        foreach(my_purple_agents[i]) begin
            amiq_dvcon_tb_vip_purple_cfg_obj proxy_purple_agent_cfg;
            proxy_purple_agent_cfg = purple_cfg(i);
            uvm_config_db#(amiq_dvcon_tb_vip_purple_cfg_obj)::set(this, $sformatf("*purpl
        end
    else if(my_purple_agents.size()==1) begin
        amiq_dvcon_tb_vip_purple_cfg_obj proxy_purple_agent_cfg;
        proxy_purple_agent_cfg = purple_cfg(0);
        uvm_config_db#(amiq_dvcon_tb_vip_purple_cfg_obj)::set(this, $sformatf("*purple_ag
    end
endfunction : configure_agents
```

Figure 5. Example of configuring the variable number of agents

Function “configure_agents()” which is being called after “cast_agents()” is responsible in our example for creating the necessary configuration objects. Because a configuration object is always necessary when an agent is created, there is no reason to dynamically create the configuration objects, because their number is fixed and tied to the number of agents. (Figure 5)

Furthermore, these configuration objects have fields that might require constraining. If they are being extended from class “amiq_ectb_object”, those fields can be controlled directly through plusargs. In this example, we assume all configuration objects are defined under pre-existing VIPs which are not based on our framework. That means that we need to have a global config object based on our framework, that assigns the relevant configuration to each VIP.

Having an object based on amiq_ectb_object brings us to our second important feature, which is dynamically defined variables. The normal workflow in UVM is that you have variables in all your relevant configuration objects which are being set from a higher layer configuration object, which itself is being assigned relevant values from the test. In our case, the test becomes a container for functionality and scenarios are completely dynamic, so these constraints from upper layer are moved outside of the simulator, in the external control layer.

```
function amiq_dvcon_tb_vip_red_cfg_obj red_cfg(int agent_id);
    amiq_dvcon_tb_vip_red_cfg_obj red_agent_cfg;
    red_agent_cfg = new("red_agent_cfg");
    red_agent_cfg.m_agent_id = agent_id;
    red_agent_cfg.m_checks_enable = env_cfg.red_vip_has_checks[agent_id];
    red_agent_cfg.m_coverage_enable = env_cfg.red_vip_has_coverage[agent_id];
    red_agent_cfg.m_is_active = env_cfg.red_vip_is_active[agent_id];
    return red_agent_cfg;
endfunction

function amiq_dvcon_tb_vip_blue_cfg_obj blue_cfg(int agent_id);
    amiq_dvcon_tb_vip_blue_cfg_obj blue_agent_cfg;
    blue_agent_cfg = new("blue_agent_cfg");
    blue_agent_cfg.m_agent_id = agent_id;
    blue_agent_cfg.m_checks_enable = env_cfg.blue_vip_has_checks[agent_id];
    blue_agent_cfg.m_coverage_enable = env_cfg.blue_vip_has_coverage[agent_id];
    blue_agent_cfg.m_is_active = env_cfg.blue_vip_is_active[agent_id];
    return blue_agent_cfg;
endfunction

function amiq_dvcon_tb_vip_purple_cfg_obj purple_cfg(int agent_id);
    amiq_dvcon_tb_vip_purple_cfg_obj purple_agent_cfg;
    purple_agent_cfg = new("purple_agent_cfg");
    purple_agent_cfg.m_agent_id = agent_id;
    purple_agent_cfg.m_checks_enable = env_cfg.purple_vip_has_checks[agent_id];
    purple_agent_cfg.m_coverage_enable = env_cfg.purple_vip_has_coverage[agent_id];
    purple_agent_cfg.m_is_active = env_cfg.purple_vip_is_active[agent_id];
    return purple_agent_cfg;
endfunction
```

Figure 6. Example of configuring VIP cfg obj based on global obj based on amiq_ectb_object

To understand how this works, we need to have a look at "…/amiq_ectb/sv/amiq_ectb_object.svh" and "…/amiq_ectb/sv/amiq_ectb_reg_functions.svh".

```
class amiq_ectb_object extends uvm_object;

    `uvm_object_utils(amiq_ectb_object)

    function new(string name="");
        super.new(name);
        register_all_vars();
    endfunction

    `include "amiq_ectb_reg_functions.svh"

endclass
```

Figure 7. amiq_ectb_object definition

In "amiq_ectb_object" we can see that a new function called "register_all_vars()" has been added to the constructor. (Figure 6) This function is defined inside

"…/amiq_ectb/sv/amiq_ectb_reg_functions.svh" and its purpose is to interrogate the plusargs database and gather values for all the variables defined using this mechanic.

```
virtual function void register_all_vars();
endfunction

function int int_reg(string my_var_name, int default_value=0);
    `uvm_info(get_name(), $sformatf("Registering field"), UVM_NONE)
    my_var_name = {get_name(), "_", my_var_name};
    `uvm_info(get_name(), $sformatf("Looking for var:%s", my_var_name),UVM_NONE)
    if(!$value$plusargs({my_var_name, "=%0d"}, int_reg)) begin
        `uvm_info(get_name(), $sformatf("Didn't find the plusarg"),UVM_NONE)
        int_reg = default_value;
    end else begin
        `uvm_info(get_name(), $sformatf("Found the plusarg"),UVM_NONE)
    end
endfunction

function bit bit_reg(string my_var_name, bit default_value=0);
    `uvm_info(get_name(), $sformatf("Registering field"), UVM_NONE)
    my_var_name = {get_name(), "_", my_var_name};
    `uvm_info(get_name(), $sformatf("Looking for var:%s", my_var_name),UVM_NONE)
    if(!$value$plusargs({my_var_name, "=%0b"}, bit_reg)) begin
        `uvm_info(get_name(), $sformatf("Didn't find the plusarg"),UVM_NONE)
        bit_reg = default_value;
    end else begin
        `uvm_info(get_name(), $sformatf("Found the plusarg"),UVM_NONE)
    end
endfunction

function string string_reg(string my_var_name, string default_value="");
    `uvm_info(get_name(), $sformatf("Registering field"), UVM_NONE)
    my_var_name = {get_name(), "_", my_var_name};
    `uvm_info(get_name(), $sformatf("Looking for var:%s", my_var_name),UVM_NONE)
    if(!$value$plusargs({my_var_name, "=%0s"}, string_reg)) begin
        `uvm_info(get_name(), $sformatf("Didn't find the plusarg"),UVM_NONE)
        string_reg = default_value;
    end else begin
        `uvm_info(get_name(), $sformatf("Found the plusarg"),UVM_NONE)
    end
endfunction
```

Figure 8. Register variables functions

As seen in Figure 8, the function "register_all_vars()" is a hook that the user populates with calls to the "*_reg()" functions. These calls allow the user to set up a name for the plusargs as well as a default value that should be used if that plusarg is not defined.

```
virtual function void register_all_vars();

   // Active/Passive
   red_vip_is_active[0] = uvm_active_passive_enum'(bit_reg("red_vip0_is_active", UVM_ACTIVE));
   red_vip_is_active[1] = uvm_active_passive_enum'(bit_reg("red_vip1_is_active", UVM_ACTIVE));
   red_vip_is_active[2] = uvm_active_passive_enum'(bit_reg("red_vip2_is_active", UVM_ACTIVE));
   red_vip_is_active[3] = uvm_active_passive_enum'(bit_reg("red_vip3_is_active", UVM_ACTIVE));
   red_vip_is_active[4] = uvm_active_passive_enum'(bit_reg("red_vip4_is_active", UVM_ACTIVE));
   red_vip_is_active[5] = uvm_active_passive_enum'(bit_reg("red_vip5_is_active", UVM_ACTIVE));
```

Figure 9. Variable registration

In Figure 9, some of the variables in the configuration object are being registered as plusargs and given the default value of "UVM_ACTIVE". The final plusargs that is going to be used is composed of the class name that instantiates the variables, aggregated with the string passed as an argument to the "*_reg()" function.

Examples on how the plusargs are defined can be found in "…/amiq_ectb/ scripts/plusarg_gen/ Verification_Environment.args" and "…/amiq_ectb/scripts/plusarg_gen/Object_definition.args".

```
+amiq_dvcon_tb_env_comp0=amiq_dvcon_tb_vip_red_agent
+amiq_dvcon_tb_env_comp0_name=red_agent
+amiq_dvcon_tb_env_comp0_no=2

+amiq_dvcon_tb_env_comp1=amiq_dvcon_tb_vip_blue_agent
+amiq_dvcon_tb_env_comp1_name=blue_agent
+amiq_dvcon_tb_env_comp1_no=1

+amiq_dvcon_tb_env_comp2=amiq_dvcon_tb_vip_purple_agent
+amiq_dvcon_tb_env_comp2_name=purple_agent
+amiq_dvcon_tb_env_comp2_no=3

+amiq_dvcon_tb_env_comp3=amiq_dvcon_tb_coverage_collector
+amiq_dvcon_tb_env_comp3_name=coverage_collector
+amiq_dvcon_tb_env_comp3_no=1
```

Figure 10. Environment definition

As seen in Figure 10, an entire environment can be defined through plusargs. The syntax used for the plusargs is broken down and explain the "**Error! Reference source not found.**" chapter. Any number of components can be defined using the syntax, as well as any number of each, together with custom names for each of them. Also, the number of each component and the name can be omitted, the framework being able to automatic add those, using the type name as name and creating only one instance of said component.

```
+amiq_dvcon_tb_env_obj0=amiq_dvcon_tb_env_cfg
+amiq_dvcon_tb_env_obj0_name=env_cfg_0
+env_cfg_0_vip0_is_active=1
+env_cfg_0_vip0_has_has_coverage=1
+env_cfg_0_vip3_is_active=1
+env_cfg_0_vip3_has_has_coverage=1
+env_cfg_0_vip5_is_active=1
+env_cfg_0_vip5_has_has_coverage=1
```

Figure 11. Dynamic object creation and variable control

Furthermore, if the "register_all_vars()" function is defined and relevant variables are added to it, all variables can be controlled as well. In Figure 11, the "env_cfg" shown earlier is instantiated in the env dynamically and it has some variables set to a particular value. Same as for the components, the syntax for the plusargs is explained in the "**Error! Reference source not found.**" chapter.

Sequence scheduling / Scenario creation

Sequence scheduling is probably the most distinct feature of this framework. We no longer think of testcases and scenarios being the same thing. Within our framework, we only have a singular testcase that creates the environment and facilitates the creation of scenarios.

Scenarios are now untied to the code and exists as a virtual component that is provided at runtime to the simulator with a distinct collection of plusargs. To understand how this is possible, we have to have a look at the base test, "amiq_ectb_test", a class that has to be extended to unlock these features.

```
class amiq_ectb_test extends uvm_test;

    // Sequence types
    string sequence_types[$];

    // Sequence names
    string sequence_names[$];

    // Sequence serial/parallel
    bit sequence_parallel[$];

    // Virtual sequencer
    uvm_sequencer virtual_sequencer;

    // Used to create components
    // Needs to be retrieved so it is instantiated globally to reduce performance hit
    uvm_factory factory;
```

Figure 12. ECTB base test and its fields

As visible in Figure 12, the test instantiates the primary queues that hold the relevant fields necessary to create a scenario. The type of the sequences that are going to be run, their names that assure their unicity and if the sequence is parallel or not. Being queues, any number of sequences can be defined to create a scenario. A virtual sequencer that has to be populated is present as well. All sequences will be

started on that sequencer. That means that casting it in the body of the sequence will offer you access to all handles inside the sequencer. (That's how relevant VIP sequencers can be accessed)

```
task run_phase(uvm_phase phase);
    super.run_phase(phase);

    if(virtual_sequencer==null) `uvm_fatal(get_name(), $sformatf("A vir

    // Retrieve the factory globally so it is available in all function
    factory = uvm_factory::get();

    // Read the plusargs for all the defined sequences and save their t
    retrieve_all_seq_type();

    // Based on the previous returned types, names and parallelism stat
    for(int i=0; i<sequence_types.size(); i++) begin
        schedule_sequence(i);
        wait_threads(i);
    end
endtask : run_phase

/**
 * @param index - Current index parsed in the type/name/parallel queues
 */
task wait_threads(int index);
    if(sequence_parallel[index] && ~sequence_parallel[index+1])
        wait fork;
endtask : wait_threads
```

Figure 13. Run phase of the amiq_ectb_test

In Figure 13 the "run_phase()" can be visible. The most important functions in the "run_phase()" are the "retrieve_all_seq_type()" and "schedule_sequence()".

"Retrieve_all_seq_type()" is the function that interrogate all the plusargs and populate the queues visible in Figure 12.

```
function string retrieve_seq_type(int index);
    string seq_index = $sformatf("seq%0d", index);
    if(!$value$plusargs({seq_index, "=%0s"}, retrieve_seq_type)) retrie
endfunction

function string retrieve_seq_name(int index);
    string seq_index = $sformatf("seq%0d_name", index);
    if(!$value$plusargs({seq_index, "=%0s"}, retrieve_seq_name)) retrie
endfunction

function bit retrieve_seq_if_parallel(int index);
    string seq_index = $sformatf("seq%0d_p", index);
    if(!$value$plusargs({seq_index, "=%0b"}, retrieve_seq_if_parallel))
endfunction
```

Figure 14. Plusarg interogation functions

```
function void retrieve_all_seq_type();
    string seq_type;
    string seq_name;
    bit seq_parallel;
    int index;

    forever begin
        // Retrieve the sequences passed as plusargs
        seq_type = retrieve_seq_type(index);
        // If no sequence name is retrieved, we break the loop
        if(seq_type == "") break;

        // If the sequence exists, retrieve its name, if that is defined
        seq_name = retrieve_seq_name(index);

        // If the name is not defined, create it based on the type and the index
        if(seq_name=="") seq_name = $sformatf("%0s_%0d", seq_type, index);

        // If the parallelism status is defined, retrieve it, otherwise it is serial
        seq_parallel = retrieve_seq_if_parallel(index);

        // Push the type, name and parallelism status
        sequence_types.push_back(seq_type);
        sequence_names.push_back(seq_name);
        sequence_parallel.push_back(seq_parallel);

        index++;
    end
endfunction
```

Figure 15. retrieve_all_seq_type() definition

The user is responsible to make sure that all types are defined correctly. The name and parallelism can be omitted, as the name will always be unique and the parallelism will be considered as serial.

```
task create_and_start_seq(string type_name, string inst_name, int index);
    uvm_object m_object;
    amiq_ectb_sequence m_sequence;

    // Create
    m_object = factory.create_object_by_name(type_name, this.get_full_name
    $cast(m_sequence, m_object);

    // If seq for this index is defined but the type is not defined, we t
    if(m_sequence == null) begin
        `uvm_fatal(get_name(), $sformatf("seq%0d is defined as a type that
    end

    // Register_all_vars
    m_sequence.register_all_vars();

    // Set the pointer to env and start sequence
    m_sequence.start(virtual_sequencer);

endtask
```

Figure 16. create_and_start_seq() definition

The “create_and_start_seq()” is responsible for creating all the sequences using the factory and start the sequence. The parallelism is handled through the wait_threads() function which will block the execution if the next sequence is marked as serial -> If the current sequence and the next one are parallel, the wait_thread won’t wait. This will be repeated until the next sequence is serial, allowing any number of parallel sequences to be scheduled at once.

A fully working extension/example of the amiq_ectb_test can be see at “/amiq_ectb/tb/tc/amiq_dvcon_tb_tc.svh”

```
virtual function void register_all_vars();
    super.register_all_vars();
    red_pkt_nr = int_reg("red_pkt_nr");
    red_agent_id = int_reg("red_agent_id");

    blue_pkt_nr = int_reg("blue_pkt_nr");
    blue_agent_id = int_reg("blue_agent_id");

    purple_pkt_nr = int_reg("purple_pkt_nr");
    purple_agent_id = int_reg("purple_agent_id");

endfunction : register_all_vars
```

Figure 17. Sequence register_all_vars

Same as for the components and objects in the environment, the sequences have the “register_all_vars()” function which allows the fields of the sequence to be set to relevant values based on the scenario that needs to be ran. An example of a sequence can be seen in “/amiq_ectb/tb/sv/seq_lib/amiq_dvcon_tb_seq0.svh”

```
red_pkt_nr      = int_reg("red_pkt_nr", 10000);
red_agent_id    = int_reg("red_agent_id", 0);

red_field0_constraints = amiq_dvcon_tb_dynamic_constraint::type_id::create
red_field1_constraints = amiq_dvcon_tb_dynamic_constraint::type_id::create
red_field2_constraints = amiq_dvcon_tb_dynamic_constraint::type_id::create
red_field0_constraints.register_all_vars();
red_field1_constraints.register_all_vars();
red_field2_constraints.register_all_vars();
```

Figure 18. Dynamic constraints added to sequence

For the purpose of automation when it comes to coverage closure and stimuli constraint optimization, we need a variable amount of constraint intervals. For this purpose, we have created a dynamic constraint object which can be reviewed at “/amiq_ectb/tb/sv/seq_lib/amiq_dvcon_tb_dynamic_constraint.svh”. This object allows us to create an indefinite amount of intervals for item constraint randomization.

```
+seq0=amiq_dvcon_tb_seq0
+seq0_name=amiq_dvcon_tb_seq0_0

+seq1=amiq_dvcon_tb_seq0
+seq1_name=amiq_dvcon_tb_seq0_1
+seq1_p=1
+amiq_dvcon_tb_seq0_1_blue_pkt_nr=3000

+seq2=amiq_dvcon_tb_seq0
+seq2_name=amiq_dvcon_tb_seq0_2
+seq2_p=1
```

Figure 19. Simple scenario based on dynamic sequence scheduling

A simple scenario created with the same type of sequence and three different instances can be seen in the figure above. In this case, for the most part, the default field values are being used with an overwrite shown for “blue_pkt_nr” in the sequence “amiq_dvcon_tb_seq0_1”.

```
class amiq_dvcon_tb_dynamic_constraint extends amiq_ectb_object;
    `uvm_object_utils(amiq_dvcon_tb_dynamic_constraint)

    int nof_intervals;
    int range_start[];
    int range_end[];
    int range_weight[];

    localparam int max_int = 2**31-1;

    // new - constructor
    function new(string name = "amiq_dvcon_tb_dynamic_constraint");
        super.new(name);
    endfunction : new

    virtual function void register_all_vars();
        super.register_all_vars();
        nof_intervals = int_reg("nof_intervals", 10);

        range_start = new[nof_intervals];
        range_end = new[nof_intervals];
        range_weight = new[nof_intervals];

        foreach(range_start[i])
            range_start[i] = int_reg($sformatf("range_start_%0d", i),
        foreach(range_end[i])
            range_end[i] = int_reg($sformatf("range_end_%0d", i),  (i
        foreach(range_weight[i])
            range_weight[i] = int_reg($sformatf("range_weight_%0d", i)
```

Figure 20. Plusarg registration in “amiq_dvcon_tb_dynamic_constraint”

Visible in Figure 20 is the registration of the dynamic intervals in the dynamic_constraint class. The name of the classes seen in Figure 18 are required to register these plusargs. Any number of intervals

can be defined by adding plusargs for +***name_of_the_dynamic_constraint_object***_range_*_***index***=***value***.

```
+seq0=amiq_dvcon_tb_seq0
+amiq_dvcon_tb_seq0_0_red_pkt_nr=10
+red_field0_constraints_nof_intervals=100
+red_field1_constraints_nof_intervals=100
+red_field2_constraints_nof_intervals=100



+red_field0_constraints_range_start_98=2147483645
+red_field0_constraints_range_end_98=2147483645
+red_field0_constraints_range_weight_98=9993
+red_field0_constraints_range_start_95=2147483642
+red_field0_constraints_range_end_95=2147483642
+red_field0_constraints_range_weight_95=9995
+red_field0_constraints_range_start_3=3
+red_field0_constraints_range_end_3=3
+red_field0_constraints_range_weight_3=9997
+red_field0_constraints_range_start_97=2147483644
+red_field0_constraints_range_end_97=2147483644
+red_field0_constraints_range_weight_97=9997
+red_field0_constraints_range_start_0=0
+red_field0_constraints_range_end_0=0
+red_field0_constraints_range_weight_0=9999
+red_field0_constraints_range_start_2=2
+red_field0_constraints_range_end_2=2
+red_field0_constraints_range_weight_2=9999
+red_field0_constraints_range_start_4=4
+red_field0_constraints_range_end_4=4
+red_field0_constraints_range_weight_4=9999
+red_field0_constraints_range_start_96=2147483643
+red_field0_constraints_range_end_96=2147483643
+red_field0_constraints_range_weight_96=9999
...
```

Figure 21. Scenario with dynamic number of

In Figure 21 a scenario based on a single sequence of type “amiq_dvcon_tb_seq0” with default name of “amiq_dvcon_tb_seq0_0” and default parallelism of “serial” is created. Separate from the sequence, the fields of the constraint object instantiated under that sequence are set to 100 intervals and each interval has a start/end value assigned for the interval, as well as a weight. How this can be used for verification ca be seen in the “Coverage Driven Constraint Adjustment” chapter, where the intervals are automatically generated based on previous regressions results.

Example of automation

Excel to plusargs converter

Using this framework, the entire testbench configuration and stimuli constraints will be described by large files containing hundreds, if not more plusargs. A different solution is required for storing and managing plusargs in a more human-readable format.

Our proposed solution is to organize the testbench and stimuli descriptions in one or more excel sheets, and then use a script(generate_plusargs.py) to generate the plusarg files from there. There are three different types of tables that are used:

1. Component description

To add a new component to the environment, the user needs to specify only where the component is instantiated, and what is its type. Optionally, the user can control the name of the component and how many instances of that particular type are created.

	A	B	C	D
1	**Parent name**	**Component Type**	**Component Name**	**Number of components**
2	amiq_dvcon_tb_env	amiq_dvcon_tb_vip_red_agent	red_agent	2
3	amiq_dvcon_tb_env	amiq_dvcon_tb_vip_blue_agent	blue_agent	1
4	amiq_dvcon_tb_env	amiq_dvcon_tb_vip_purple_agent	purple_agent	3
5	amiq_dvcon_tb_env	amiq_dvcon_tb_coverage_collector	coverage_collector	1

Figure 22. Excel spreadsheet for creating an environment

The excel sheet from Figure 22 will be converted to the plusargs from Figure 10.

2. Object description

An object can be added in any component that extends amiq_ectb_component or amiq_ectb_environment. To be able to generate the plusargs, the table definition in Figure 23 is required. On the first row, the user defines where the object is instantiated, what type it is and the name with which it will be created. On the next rows, the user will list the fields from the object that will be updated.

	A	B	C	D	E
1	**Parent name**	**Object Type**	**Object Name**	**Field**	**Value**
2	amiq_dvcon_tb_env	amiq_dvcon_tb_env_cfg	env_cfg_0	vip0_is_active	1
3				vip0_has_checks	0
4				vip0_has_has_coverage	1
5				vip3_is_active	1
6				vip3_has_checks	0
7				vip3_has_has_coverage	1
8				vip5_is_active	1
9				vip5_has_checks	0
10				vip5_has_has_coverage	1

Figure 23. Introducing an env_cfg under amiq_dvcon_tb_env with field values set to certain values

3. Sequence description

A sequence is defined similarly to an object, the difference are that the parent is not specified in the table, as all sequences are created in the test, and that there is an extra “Run in parallel” column, to be

able to define how the sequences will be started (more details in Sequence scheduling / Scenario creation section)

	A	B	C	D	E
1	Sequence type	Sequence name	Field	Value	Run in parallel (YES/NO)
2	amiq_dvcon_tb_seq0	amiq_dvcon_tb_seq0_0	red_field0_constraints_range_start_0	0	NO
3			red_field0_constraints_range_end_0	1024	
4			red_field0_constraints_range_weight_0	100	
5			blue_pkt_nr	0	
6			purple_pkt_nr	0	
7	amiq_dvcon_tb_seq0	amiq_dvcon_tb_seq0_1	red_pkt_nr	0	YES
8			blue_pkt_nr	3000	
9			blue_agent_id	0	
10			purple_pkt_nr	0	
11	amiq_dvcon_tb_seq0	amiq_dvcon_tb_seq0_2	red_pkt_nr	0	YES
12			blue_pkt_nr	0	
13			purple_pkt_nr	1500	
14			purple_field0_constraints_range_start_0	0	
15			purple_field0_constraints_range_end_0	127	
16			purple_field0_constraints_range_weight_0	50	
17			purple_field0_constraints_range_start_1	128	
18			purple_field0_constraints_range_end_1	512	
19			purple_field0_constraints_range_weight_1	50	

Figure 24 Scenario definition through an excel spreadsheet

The *generate_plusargs.py* script can have as an input multiple .xlsx files, each of them having multiple sheets used to describe testbench elements (components/objects/sequences). The script will output a plusarg file for each sheet with the naming convention *<sheet_name>.args*.

Excel example: */scripts/plusarg_gen/tb_params.xlsx*

Run example: *python3.6 generate_plusargs.py tb_params.xlsx*

Post-session script - Coverage Driven Constraint Adjustments

One of the main benefits of using the ectb framework is that the plusargs represent a set of clearly defined parameters that can control the behavior of the testbench. Updating these parameters would allow the test environment to stimulate the DUT in various ways, from randomly exploring and bug-hunting to targeting specific scenarios.

The post-session script developed for the example TB has the goal to automatically fill-up the coverage based on the results from previous regressions. The flow is described in Figure 13 has the following steps:

1. Run a regression
2. Interrogate the database using Coverage Lens and create a local coverage database in JSON format
3. Merge the JSON with previous coverage results
4. Check end condition
    a. If a certain number of regressions sessions have been run or the coverage goal has been hit, then we are Done

b. Otherwise generate new set of plusargs

5. Start a new regression with the newly generated parameters.

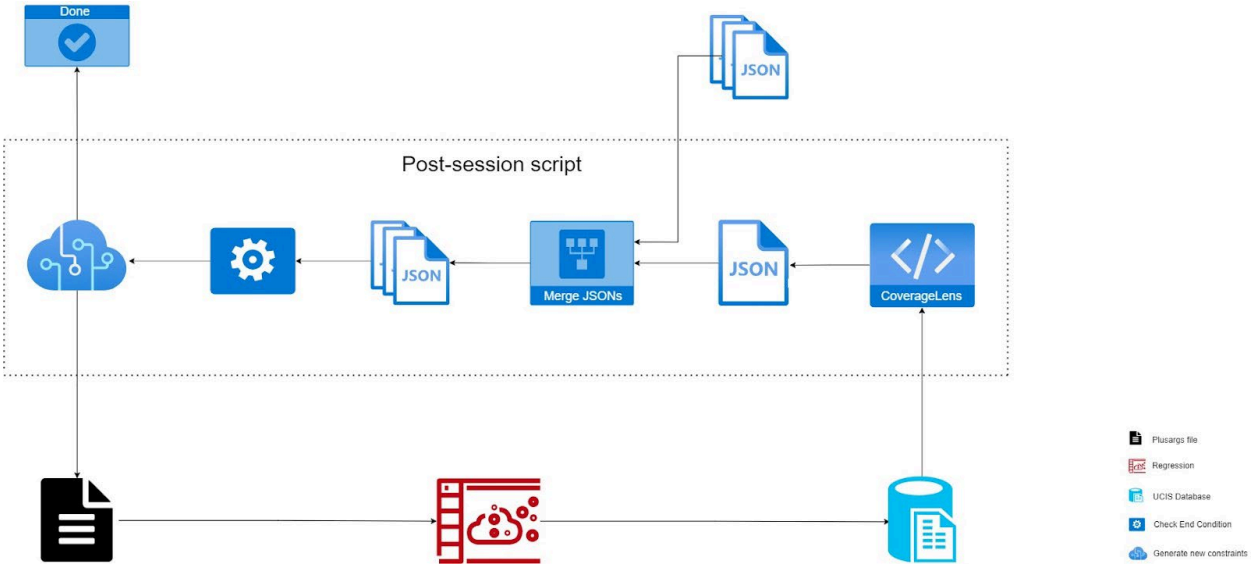

Figure 25. Post-session script for automatic coverage optimization - In-depth view

To be able to take advantage of this flow, the user needs to update two files in /scripts/post_session:

- post_session.config -> from here the post_session.py is taking external parameters such as maximum number of runs, path to .vsif, path to store coverage database,etc
- update_parameters.py -> In post_session.py we call *update_seq_parameters*(*coverage_db*) which should be implemented by the user. This function will contain the algorithm that translates the merged coverage results provided by the post-session script, to new stimuli (new plusarg files)

Plusarg Syntax

1. Component definition

+parent_name_comp#=type_of_component

- This is mandatory for defining a dynamic component. Any type of component can be used. Upon running the build_phase of an environment extended from amiq_ectb_environment, one or “number_of_instances” components will be present in the components[$] queue. They can be then $cast’ed based on their type_name and the relevant instance can be singled out based on the “name_of_instances” or based on the generic name based on type_name and the index of the instance.

+parent_name_comp#_no=number_of_instances

- Optional field that can tell the framework to create “number_of_instances” agents of the comp# type defined with the previous plusargs. If not defined, default will be one, hence only one instance will be created.

+parent_name_comp#_name=name_of_instances

- Optional field that can tell the framework what name to be used for the created instances of type signaled with the first plusarg (*_comp#=) in this list. If multiple instances are to be created, as indicated by plusargs “comp#_no”, each instance name will have the index appended as such: *name*_*index*

2. Object definition

+parent_name_obj#=type_of_object

- Similar to the component, but used for the creation of objects

+parent_name_obj#_no=number_of_instances

- Similar to the component, but used for the creation of objects

+parent_name_obj#_name=name_of_instances

- Similar to the component, but used for the creation of objects

3. Sequence scheduling

+seq#=type_of_sequence

- Similar to how comp/obj works, here we have just seq*index* equal to the type of the current sequence.

+seq#_name=name_of_sequence

- Similar to how comp/obj works, for each seq*index* we can set a name. Here each sequence defined has only one instance

+seq#_p=0/1

- To be able to create different sequence trees, for each sequence we can specify its parallelism status. If set to 0, the current sequence is serial, which means that it will start when all previous parallel sequences have ended. If this is parallel, the next one will be checked as well and so on subsequently until all consecutive parallel sequences have started.

4. Registered field assignment

+parent_name_field_name=value

- Any field that has been registered in the “register_all_vars() or any other function that gets called before the field is used, using the “*_reg()” function, can be given an arbitrary value.

Legend:

parent_name = Has to be a valid component which is extended from amiq_ectb_component or amiq_ectb_environment. In case of a field registration, can be any object, component or object.

= Has to be a valid incremental index for each parent. The first comp/obj that will be created for a given parent has to be comp0 or obj0. The next one will be 1, and so forth.

type_of_component/object/sequence = A valid type of component/object/sequence that was registered with the factory. This will be used via factory to create the desired component/object.

number_of_instances = Has to be a valid integer that will result in as many instances of the current type to be created. The current type is the one specified by comp/obj*index* with the same *index* as comp/obj*index*no=…

name_of_instances/sequence = Has to be a valid string that will be used as the base name for the instance at the current index of comp/obj*index*. If multiple instances are defined, "_number" will be appended at the end of string for each component. Number will be between 0 and "number_of_instances – 1". For sequences, only one instance will be created for each seq*index* plusargs with the desired name.

field_name = Has to be a valid field name set for a valid via the "*_reg" function inside a object/component or sequence that extends from its relevant ectb base class.

Value = The relevant value for the type of "field" used (Ex: string, int, bit)

Important note: To uniquely identify a field in a particular container, it is important to have unique names for all components/objects/sequences. If this is not the case, one plusarg definition will update the fields in all the containers with the same name.