Five Ways to Make Your Specman Environment More Reusable and Configurable

A Simple Guide With Code Examples

Stefan Sljukic, Veriest Solutions, Belgrade, Serbia (stefans@veriests.com)

Nikola Knezevic, Veriest Solutions, Belgrade, Serbia (nikolakn@veriests.com)

Filip Dojcinovic, Veriest Solutions, Belgrade, Serbia (filipd@veriests.com)

***Abstract* — This paper will present some of the approaches and solutions for creating a more reusable and configurable Specman/e environment. The paper will include the code examples based on the "hands-on" experience and conclusions regarding how we can prevent and solve some of the problems in the process of creating a good Specman/e environment. The paper will not cover all the necessary properties of a good Specman/e environment, but rather will focus on five specific ways to make your Specman/e environment more robust.**

Keywords* — *Specman/e; UVMe; Specman macros; environment configuration; reusabillity; scaleability

I. Introduction

A good verification environment must be one of the most important goals of any verification engineer. It does not matter in which programing language you work it is always necessary to make a robust verification environment which, not only covers and checks every feature but is also easy to read and understand.

How many times have you encountered a problem which needed a good deal of creativity to be solved? Maybe you found a great solution for some of the problems that could be useful to other engineers with the same issue. On the other hand, maybe your colleague has an even better solution for the same problem. It is clear that we need to share our knowledge in order to grow as engineers and be able to tackle a growing number of complex problems that await us. This paper will do exactly that. We will share some of our solutions for not so uncommon problems that any verification engineer could find helpful. Given the fact that we are not addressing the basics of verification or the Specman/e language, the one reading this paper should be familiar with the verification methodologies and E language to fully utilize the solutions and code examples presented.

II. One Macro To Connect Them All

Specman/e macros are a powerful tool. They wield the power much greater than macros in System Verilog. But one must use them carefully. With great power comes great responsibility, and in the sense of verification environments, the tool that can help it be more reusable can also make it less readable and harder to debug. Thus, one of the safe ways to use them is to connect the ports of the agents that appear multiple times in the environment. Let us say that the RTL has several inputs/outputs of the same protocol. Depending on the protocol, it can have many ports that need their HDL paths to be defined. In this case, using Specman/e macros is a great way to keep your line count lower, and it is easily upgradable. The life of a block in the RTL can be dynamic while in development and we are witnesses to a lot of changes until the tape out. So, keeping that in mind can save us a lot of time in the future if adding or removing agents is needed. We will give an example of a good macro that is readable, easily scalable, and easy to debug.

Given the fact that the Specman/e macros are used in numerous different ways, and for many different purposes we will focus on a simpler part of the usage spectrum. Many companies that are using Specman/e have developed almost a new programing language using the Specman/e macros which only proves how powerful they are. On some projects, managers will not even allow you to implement complex macros as it complicates the knowledge transfer when blocks change owners. One of the best and desirable places where you can use a macro is the connection of UVC interface ports. It will make your life a bit easier.

Let us assume that you have multiple instances of the same UVC in your environment. For example, you have several interfaces of the same kind. It can be painstaking to connect all the ports to their respective RTL counterparts but even more painstaking if you must do it multiple times for the same type of interface. In Example 1 you can see the one such macro that is used for connecting an APB UVC to the RTL signals.

Example 1 Simple Connection Macro Definition

```
define <connect_apb_uvc'statement> "CONNECT_APB -sub_type <sub_type'type> -hdl_path <hdl_path'exp> -
prefix <prefix'exp> -suffix <suffix'exp> " as {

        extend apb_types_t:[<sub_type'type>];

        extend <sub_type'type> apb_ports_u {
        keep hdl_path() == <hdl_path'exp>;

        // Connection of the ports. One does not need to connect all the ports, but only the used
        ones.
        keep apb_pclk_p.hdl_path() == append(<prefix'exp>, "pclk", <suffix'exp>);
        keep apb_presetn_p.hdl_path() == append(<prefix'exp>, "preset", <suffix'exp>);
        keep apb_paddr_p.hdl_path() == append(<prefix'exp>, "paddr", <suffix'exp>);
        keep apb_psel_p.hdl_path() == append(<prefix'exp>, "psel", <suffix'exp>);
        keep apb_penable_p.hdl_path() == append(<prefix'exp>, "penable", <suffix'exp>);
        keep apb_pwrite_p.hdl_path() == append(<prefix'exp>, "pwrite", <suffix'exp>);
        keep apb_pwdata_p.hdl_path() == append(<prefix'exp>, "pwdata", <suffix'exp>);
        keep apb_pready_p.hdl_path() == append(<prefix'exp>, "pready", <suffix'exp>);
        keep apb_prdata_p.hdl_path() == append(<prefix'exp>, "prdata", <suffix'exp>);
        keep apb_psel_p.hdl_path() == append(<prefix'exp>, "preset", <suffix'exp>);

        };
};
// The example of using the Macro
CONNECT_APB -sub_type APB_SLV_0 -hdl_path "top.dut.sub_block_0" -prefix "sub_blk_0_"      -suffix
"_slv";
CONNECT_APB -sub_type APB_SLV_1 -hdl_path "top.dut.sub_block_1" -prefix "sub_blk_1_"      -suffix
"_mst";
```

In the example above you can see the definition of the simple connection macro. As an argument, we pass sub_type of the UVC, hdl_path to the block in which the APB interface is located and prefix and suffix to be able to connect the interfaces of any block in the verification environment. As a prefix, we can have the name of the block or the name of the master or the slave connected to the interface, or any other prefix or suffix determined in the project.

We first extend the apb_types_t to add the subtype to the respective UVC types. Then extend the ports of that type and constrain the hdl_path of the ports unit to have the hdl_path passed as an argument of this macro. One can add more arguments to this example that will address the specific UVC Keep in mind that it should be simple as possible. You want it to be readable and easy to debug.

As you can see, when you implement the macro, its usage is simple. You just invoke the macro as you defined it with its arguments and that is it. You can do this for all the UVCs you have in your environment. If you find an issue or the nomenclature is changed in the middle of the project, you can simply change it in one place and not go over each place you extended the subtype of the unit.

III. One Test To Test Them All

We all know the power of having the base test from which all others should inherit. It is a good practice, and it saves a lot of time for debugging and regression cleaning. But if we are talking about the registers test, we can have one test to test all the register files in the project. This way we can be sure that every register file in RTL has been tested the same way and for the same features. For each of the blocks, we can only specify register name and we can just add the test to our regression. This chapter will provide the code example of such a test.

When creating any good test plan, for any block, you should always plan to have one base sequence that will be inherited by all other tests. In most cases that sequence will be the most complex and all the other tests will contain just the constraints needed for the specific scenario. This approach is desirable as it avoids duplication of code and is more reusable. You can easily add the new scenario by simply adding some sequence constraints that should hit the specific scenario that you defined in your test plan. Keeping that in mind, there is one type of test that can hugely benefit from that approach – the registers test. On any given project you can have many registers in numerous blocks. In any block that contains a registers file, you need to have a registers test.

Many engineers will probably have a different way of testing the registers that can lead to some bugs being overlooked. Registers in your project should be tested the same way, and you can devote some time to create a basic registers test that will be used in all the blocks in the project. In Example 2 you can see a simple outline of how one such test can look like.

Example 2 Basic Registers Test

```
extend BASIC_REG vr_ad_sequence {

        !reset_check_seq: RESET_CHECK vr_ad_sequence;
        !read_write_read_seq: READ_WRITE_READ vr_ad_sequence;
        !irq_check_seq: IRQ_CHECK vr_ad_sequence;
        !shadow_reg_check_seq: SHADOW_REG_CHECK vr_ad_sequence;

        reg_file_name: string;

        rf: vr_ad_reg_file;

        reset_hdl_path_list: list of string;


        body()@driver.clock is only {

                rf = driver.addr_map.reg_file_list.first(.name == reg_file_name);
                assert rf != NULL else
                        error("Register file named ",reg_file_name," is not found!");

                // Checking the reset values of the registers
                do reset_check_seq keeping {
                        .driver == driver;
                        .reg_file == rf;
                };

                // Reading and writing to registers
                // Also checking the accesses to RO, WO, and RW registers.
                do read_write_read_seq keeping {
                        .driver == driver;
                        .reg_file == rf;
                };

                // The hook tcm which can be extended by inheriting tests
                do_pre_reset_tcm();

                // We do the reset
                do_reset_tcm();

                // The hook tcm which can be extended by inheriting tests
                do_post_reset_tcm();

                // Again, we check the reset values
                do reset_check_seq keeping {
                        .driver == driver;
                        .reg_file == rf;
                };

                // You can add the other checking sequence to the flow.
                do irq_check_seq keeping {
                        .driver == driver;
                        .reg_file == rf;
                };
                do shadow_reg_check_seq keeping {
                        .driver == driver;
                        .reg_file == rf;
                };
        };
};
```

As you can see in the example, we have several subsequences in our registers test. We will not go deep into those sequences but rather have them as placeholders for the actions described by their names. As shown, we first check the reset values of the registers, then check the accesses with read-write-read sequence, and so on. We also have some hook methods that can allow us to implement more checks or any other type of actions.

Thus, when you want to use it, you will need only to specify the name of the reg file so we can have the register test for the block implemented. Example 3 is showing the usage of the basic registers test.

Example 3 Usage of Basic Registers Sequence

```
extend MAIN sequence_s {
        !basic_reg_seq: BASIC_REG vr_ad_sequence;

        reg_file_names_list: list of string;
        body()@driver.clock is only {
                reg_file_names_list.add("reg_file_1", "reg_file_2");

                // If you want to check specific regfiles in reg model
                for each (rf_name) in reg_file_names_list {
                        do basic_reg_seq keeping {
                                .driver == driver.p_env.rsd;
                                .reg_file_name == rf_name;
                                -- The reset of the block in which reg file is located
                                .reset_hdl_path_list == {"reset_n";};
                        };
                };

                // If you want to test all the regfiles in your model you can do it easily
                for each (regfile) in driver.p_env.addr_map.regfiles {
                        do basic_reg_seq keeping {
                                .driver == driver.p_env.rsd;
                                .reg_file_name == regfile.name;
                                -- if you want to reset the directly the reg files inside
                        };
                };
        };
};
```

Take into consideration that you need to specify the hdl_path for each register file that you use, and you will be able to access it in your test. You can test all the reg files in your test just by using the power of vr_ad methods. There are several vr_ad methods that we would like to point out that can be helpful in the development of one such test which are presented in Example 4.

Example 4 Some Useful vr_ad Methods

```
register_list: list of vr_ad_reg;
registers_list = reg_file.get_all_regs(); -- this returns all regs of the reg file

-- this is the list of field attributes of the specific register_list
register_list[index].get_static_info().fields_info

// usage example of the above vr_ad method
for each (attr) in reg.get_static_info().fields_info{
        if(attr.fld_mask == RO){
                -- do some action
        };
        if(attr.fld_mask == WO){
                -- do some action
        };
};

register_list[index].get_backdoor_paths()

register_list[index].get_reg_by_kind()
```

IV. HELPER UNITS COME TO RESCUE

When you have a block that has a lot of inputs of the same kind, and it has a lot of versions with a different number of inputs, it can cause a lot of headaches. We will group all the events in one unit. Then we will make a list of helper units and it will have a configurable size depending on the version. Moreover, it can be accessed both from sequences and reference model. You can also use a helper unit for the connection of multiple agents to the reference model. This chapter will provide simple code examples of such units and how they can be placed.

Let us assume that you have multiple sub-blocks in your design that have some FIFOs in them. You want to have some action flow in your reference model when FIFO flush happens. To make matters worse, your design can have a different number of those blocks determined by a particular parameter. Helper classes can offer an elegant solution in this situation. Look at Example 5.

Example 5 Helper Class for Events

```
extend dut_env_u {
        cfg: dut_cfg_u is instance;
        ref_model: dut_ref_model_u is instance;

        seq_driver: dut_seq_driver_u is instance;

        //List of helper units
        sub_block_event_wrappers: list of sub_block_event_wrapper_u;
        keep sub_block_event_wrappers.size() == cfg.num_of_sub_blocks;
        keep for each in sub_block_event_wrappers {
                sub_block_event_wrappers[index].idx == index;
        };

        // connect the pointers of each helper unit in the list with ref model, and index it.
        connect_pointers() is also {
                ref_model.p_sub_b_event_wrapp.resize(cfg.num_of_sub_blocks);
                for each in sub_block_event_wrappers {
                        sub_block_event_wrappers[index].idx = index;
                        sub_block_event_wrappers[index].p_ref_model = ref_model;
                        ref_model.p_sub_b_event_wrapp[index] = sub_block_event_wrappers[index];
                        seq_driver.p_sub_b_event_wrapp[index] = sub_block_event_wrappers[index];
                };
        };
};

extend dut_ref_model_u {
        do_fifo_flush_flow(idx: uint)@clk_r is {
                // some action
        };

};

extend sub_block_event_wrapper_u {
        idx: uint;

        fifo_flush_p: in simple_port of bit is instance;
        fifo_flush_p.hdl_path() == appendf("top_tb.sub_blk_inst[%d].fifo_flush", idx);

        event e_fifo_flush is rise(fifo_flush_p$)@clk_r;
        on e_fifo_flush {
                p_ref_model.do_fifo_flush_flow(idx);
        };
};
```

In Example 5 you can see how the helper unit has an event that is triggered by the FIFO flush from the RTL. On that event, we invoke the method from the reference model and pass the index of the helper unit as the method argument. This way we have the same flow for every instance of the sub-block FIFO. All we need to do to change the number of helper classes is to change the field (num_of_sub_blocks) value in the configuration unit. Also, this is useful if you use your environment in a higher-level environment where you have multiple instances of the same block with different parameters. Just configure the field in the configuration class and you are good to go.

Moreover, you can also use a similar idea to have multiple EVCs to report to the reference model. You can have a helper unit with the interface port in it and the write method. Then you implement a method that does some action in the reference model and has an index as an argument. You then connect the analysis ports of the EVC to the input analysis port of the helper unit and in the write method of the helper class you just call the method you implemented in the reference model and pass the index as an argument, like in Example 5.

Example 6 shows the use of the “context” option when you have the list of input analysis ports. This changes the declaration of the write method and adds another argument to it – port. As you can see, you will get the information from which port the transaction came and use it to find the index of the port. You can later use the index as an input to other methods.

Example 6 Using Context in Input Analysis Ports

```
extend dut_ref_model_u {
        new_input_trans_p: list of in interface_port of tlm_analysis of input_agent_item_s
                using suffix= _new_input_item , context is instance;
        keep new_input_trans_p.size() == p_cfg.num_of_inputs;
        keep for each in new_input_trans_p {
                soft bind(new_input_trans[index], empty);
        };

        write_new_input_item(item: input_agent_item_s, port: any_interface_port) is{
                var v_port_index := new_input_trans_p.first_index(it == port);
                // continue with action
        };
};
```

V. EXPRESS YOURSELF WITH HDL EXPRESSION

Let us say your RTL has a simple interface, but you do not have an agent for it. You have an agent for the other, slightly different interface and you want to use it and not lose time in creating a new agent from the scratch. You have an idea! Let us concatenate the several control signals that are missing from the agent port list with the data port, and it will be easy to parse them in the reference model. With HDL expression you can do concatenation of several signals as well as the other manipulations and not change the Verilog code. The HDL expressions give your environment more reusability, so they can be used at the bigger block-level or system level. This chapter will provide code examples of the use of HDL expressions for different purposes.

Example 7 contains several possible cases for HDL expression usage which we found particularly useful. If you want to use a UVC you already have and not create a new one for some simple interface, you can resort to just concatenating the signals into one that you have in your ports. After you declare the range of the specific signal you just concatenate the signals as shown in the example. It is better to use the HDL expression for that purpose than to make a new signal in your testbench and concatenate it in Verilog directly cause this way you can use it in a higher-level environment more easily. One can also use HDL expressions to create “smarter” ports. You can use Verilog operators to make a valid clock signal for example.

Example 7 HDL Expressions Usage

```
extend dut_ports_u {
        // Concatenation of several signals in one data port
        data_p : inout simple_port of uint is instance;
        keep bind(data_p, external);

        keep data.hdl_path() == "~/top.dut ";
        keep data.declared_range[25:0]; // data[15:0]; data_id[7:0]; data_first [0:0]; data_last[0:0];
        keep data.hdl_expression() == "~/{<p>.data, <p>.data_id, <p>.data_first, <p>.data_last}";

        // Using verilog operators to create in hdl expression
        valid_clock_p: inout simple_port of bit is instance;
        keep bind(valid_clock_p, external);

        keep valid_clock_p.hdl_path() == "~/top.dut";
        keep valid_clock_p.hdl_expression() == "<p>.clk & <p>.clk_en";

        // when a signal we want to connect to is a vector or matrix
        method_type array_t(i:int);

        signal_array_p : inout siple_port(array_t) of bit is  instance;
        keep bind (signal_array_p , external);
        keep signal_array_p.hdl_path() == "~/top.dut ";
        keep signal_array_p.hdl_expression() == "<p>.array[<i>]";

        // matrix
        method_type matrix_t(i:int, j:int);

        signal_matrix_p : inout siple_port(matrix_t) of bit is  instance;
        keep bind (signal_matrix_p , external);
        keep signal_matrix_p.hdl_path() == "~/top.dut";
        keep signal_matrix_p.hdl_expression() == "<p>.matrix[<i>][<j>]";

};
        // Usage
        signal_array_p.index(0)$ = 0;
        signal_matrix_p.index(0,0)$ = 0;
```

One of the great ways to use HDL expression is to have indexed ports, and with HDL expression you will enable access to members of a signal array or even a matrix

VI. Small Things Sometimes Mean A Whole Lot

While the previous chapters cover some specific ways to make your environment better, this one will gather some of the small things that you can do for your environment. We will provide the code examples of several solutions for specific problems.

A. Reflect on Your Code

The use of the reflection interface of Specman/e is not typical in the life of a test developer. The introspection, how it is also called, is not even meant to be used by a coder developing Specman/e environment. Even though it is mostly used by tool developers and advanced programmers, there is an interesting way for a verification engineer to use the reflection interface. You can check if some method or event is extended and implemented. Maybe you are developing a basic unit to be used by other units and by other engineers. And you have a method which you want to ensure is implemented – an introspection of your environment. Example 8 is showing that.

Example 8 Usage of the Reflection Interface

```
unit my_unit_u {
    my_method() is empty;
    event my_event;
    run() is also {
           check_layers();
    };

    // This method will check that method is extended/implemented and event is connected.
    check_layers() is {
         var struct_rf : rf_struct = rf_manager.get_struct_by_name("monitor_u");
         var method_rf : rf_method = struct_rf.get_method("my_method");
         var event_rf  : rf_event  = struct_rf.get_event("my_event");

        assert( method_rf.get_layers().size() > 1);
        assert( event_rf.get_layers().size() > 1);
    };
};

// Without extending my method monitor_u, assertion will fail.
extend my_unit_u {
    event my_event is only @other_event;
    my_method() is {
         // some action
    };
};
```

B. Macros for Coverage

Given the fact that we discussed the power of macros, we do not need to mention how powerful they are. Macros “as computed” can create a whole new language within the e language. One must be careful when using them because they are harder to debug but shouldn’t be too afraid to use them in a controlled manner. Thus, coverage is a good place that can be used in a lot of different ways. In Example 9 we showed one of them. It shows how to make a macro that can be used to cover each bit in a signal.

Example 9 Macro "Computed As" for Coverage

```
// easily cover each bit in a signal using macro
define <bit_cover'strct_member>
"bit_cover <e_name'exp> <num> <cov_name'name> <cov_data'name>" as computed {
    var res: list of string;
    res.add(appendf("cover item_cover_ev is also {", <e_name'exp>));
    for i from 0 to <num>.as_a(uint)-1 {
        res.add(appendf("item %s_%d : bit = %s[%d:%d];\n",
            <cov_name'exp> , i , <cov_data'exp>, i, i));
    };
    result = str_join(res, " ");
};
// usage
bit_cover item_done_e 32 data_signal, smp.data$;
```

C. *Emit Events on Register Access*

Emitting events on register access is a good way of making your verification environment more efficient. Example 10 is showing how you can do it. It will again be shown as a part of the macro because, as you already noticed, we like to wield the power of macros.

Of course, you do not have to use it in macros, but we wanted to encompass several different areas that we mentioned in this paper in this example. We have several pairs of instances of the reg file, we also have different versions for TX and RX, and we use the helper class for events. At the bottom, we have the usage example of the macro. This macro can also contain several different registers extensions to improve the efficiency of the code.

Example 10 Emitting Events on Register Access

```
define   <extend_reg_emmit_event_on_write'statement>   "EXTEND_SUB_BLOCK_REGS_EMIT_EVENT_ON_ACCESS   of
instance <inst_num'num>" as {

   extend REG_SPECIAL_<inst_num'num> vr_ad_reg {
      post_access(direction: vr_ad_rw_t) is also {
         if(direction == write) {
            for each (parent) in get_parents() {
               if parent is a vr_ad_reg_file(rf) {
                  if(rf.name == "tx_subblock") {
                     emit p_event_wrapper[<inst_num'num>].e_tx_special_wr_access;
                  }else if(rf.name == "rx_subblock"){
                     emit p_event_wrapper[<inst_num'num>].e_tx_special_wr_access;
                  };
               };
            };
         };
      };
   };
};

EXTEND_SUB_BLOCK_REGS_EMIT_EVENT_ON_ACCESS of instance 0;
EXTEND_SUB_BLOCK_REGS_EMIT_EVENT_ON_ACCESS of instance 1;
```

VII. Conclusions

As a verification engineer, one must be able to adapt to the new tool or language and to be able to work efficiently in any of them. On the other hand, this is not an easy task, so this paper aims to help an engineer in the process of creating a good and robust verification environment. The solutions presented are not the only possible solutions to verification problems presented here but will increase the quality of any given environment in the sense of readability, scalability, and reusability. This paper is aiming to encourage verification engineers to share their knowledge and learn from other people's experiences. It can give a different perspective to verification engineers and can bring to light the ideas to easily solve some of the verification challenges.