2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

How creativity kills reuse - A modern take on UVM/SV TB architectures

Andrei Vintila, AMIQ Consulting, Bucharest, Romania (*andrei.vintila@amiq.com*)
Sergiu Duda, AMIQ Consulting, Bucharest, Romania (*sergiu.duda@amiq.com*)

***Abstract*—The purpose of this paper is to set the groundwork and guidelines for a revised UVM/SV architecture and a way of working that would tick the following boxes:**

- **Provide a general testbench (TB) architecture that has a fixed structure and can be adapted to any verification requirements, no matter the scale, nature or complexity of the verified design-under-test (DUT)**
- **Enforce the reusability of the TB by constraining the usage of user defined types to allow scaling down as well as scaling up the number of verification components via the factory (string-based definitions instead of hard types)**
- **Decouple the stimulus generation from the environment and use the test as a translation layer for runtime-defined stimulus (modular sequences and adaptable constraints settled at runtime)**
- **Automate the process of constraint adjustments, sequence instantiation, and environment scaling**

I. Goals, Ideas, and Problems

The essential characteristic of a TB architecture that allows any other TB to be adapted to the new infrastructure is to be as type agnostic as possible. The concept of object creation in UVM is intermediated by the Factory component. However, along the years I've seen that the factory is often underused, its main usage being the override function. This greatly limits the capacity of scaling TBs, as every single, clearly defined type, brings with it a compilation dependency in the verification environment (VE). This is not a limitation of the UVM standard, but rather an implementation choice that has been widely used due to its simplicity. Therefore, it’s the simple change to object creation by name that will allow the environment to scale freely without code modifications.

This means that the environment can have a flat structure in which all components of a certain nature (Objects, Components, uvm_reg_blocks) can be grouped together in arrays and selectively built in a loop based on the types that are available in the Factory. If for a certain application of this particular TB, one type is not available (let’s assume that one or more VIPs are not utilized), the creation of those instances can be skipped seamlessly when the type is not present in the Factory.

As you can probably guess, this would create a problem if the sequences would be chained together and instantiated in the tests as the general UVM usage dictates. Their dependency on certain VIPs would mean that the entire hierarchical structure of the test would be affected. This represents another implementation pattern that is not a UVM limitation but has rather been a widely spread practice. The sequences don’t have to be directly instantiated in the test and the upper layer of the sequence can be completely dynamic, either by reading the types from a file or by having them provided in the command line with plusargs or a combination of both.

This opens the door for data post-processing. Sequence types and even constraints on those sequences can be modified without touching the code. The results of a verification effort are not dictated by a test, but by one or more regressions. The analysis of regression results is one of the most time-consuming parts of verification. By moving the constraints, that would otherwise be manually tinkered with, out of SV and into a text format, it gives automation a chance to make a significant speed-up in this effort. Both the compilation time, and implicitly the debug time, will be lowered if there are no dependencies between the code and the stimuli, leading to an important development cycle boost. Sequences, flags and constraints can be automatically modified based on external scripts that are aware of results like pass rate and coverage.

II. Motivation and Stereotypes

A. Testbench Complexity does not Have a Linear Increase Compared to RTL

An important aspect of IC development is that verification has always been more complex compared to the design counterpart. The different angles that have to be covered, the redundancy and the statistical approach make verification effort larger and slower. However, those are the founding bricks of the current methodologies. While manageable for smaller projects and for incremental changes, developing a large-scale ASIC from scratch is a monumental effort and something that even giant companies would steer away from.

The main motivation that drives this decision is the fact that verification tends to have, for some RTL patterns, an exponential increase in complexity with synthesizable logic. This makes sense because the chip is supposed to cover a particular scenario that should happen, while the associated verification process has to cover use-cases that should happen and scenarios that should not.

Also, the approach of constrained-random and scenario-based verification sounds brilliantly in theory. We use statistics to exercise logic, we steer a good chunk of the computing logic towards stressful stimuli and corner cases, and we touch interesting use cases with specific scenarios. So, if the recipe is there, why does it take so long to start from scratch and why the verification environment doesn't handle every RTL change automatically?

My two cents worth on the topic is that hardware verification sits somewhere in the limbo between software and hardware. The requirements cannot be relaxed enough to have a straight hardware mindset and the languages and tools are lagging behind the software development practices. That's where we have to adjust.

B. Repetition is not Always the Mother of Learning

Looking at myself and those around me, I can definitely make the statement that iterative work on coverage closure and debugging is a slow and insidious killer, much like cholesterol. Also, it's always the same mentality: I have to keep doing the same thing now to reach the deadline and I'm going to redesign the tedious and problematic parts after the current project. And you just keep on living life from a deadline to another.

However, in some cases, that redesign really does happen and given the open nature of the UVM/SV universe, the possibilities are vast. So vast in fact, that the chances of finding an optimal implementation are slim to none. The natural way of solving problems in verification is to add new functions on top of the existing TB. That's why it works on smaller projects, because the complexity is low enough to be able to grasp the implications of each functional addition. The problem arises when the project is bigger, because past a threshold, the increase in complexity is much higher than the functionality. That's the point where iterative changes and updates become a nuisance and the return of investment on effort looks thin.

C. The Road from Idea to Implementation

Another fundamental choice that we got used to is the implementation of hierarchical testcases. A base test usually comprises general functionality and is extended to add or switch on/off certain bits of functionality. That functionality is also defined in hierarchical sequences, a base type that keeps handles to relevant objects and general functionality, while extensions implement specific scenario pieces.

Therefore, when new functionality is to be added, you have to revisit the code and adapt in such a way that the new features fit right in. The place that is most suited for these updates is always at the latitude of the engineer. Giving the same problem to two engineers will usually result in two different solutions. While this is marvelous when doing art, technical work thrives on templates and methodologies.

Everybody would agree that the best TB to work with will always be the one you understand best. In that regard, possible implementation variations are always increasing the complexity of our work. To successfully increase the ease of use, the employed methodology should enforce or greatly favor a fixed architecture. This way, migrating between different TBs and projects should be possible with the least amount of ramp-up. Also, when the architecture is fixed, the steps in developing something from scratch should be easier to estimate, making its prospect easier to digest.

D. *Programming code vs Human readable text*

Everybody doing verification has become accustomed to the term: "control knobs". The problem is that the propagation of these control knobs is implementation specific. Also, the control knobs are usually randomizable fields in the tests or virtual sequences. The collection of testcases is basically a list of permutations of these control knobs.

Our novel approach on implementing the control knobs is to take them out of the SV code. When having implicit randomization blocks that vary from one test to the other, the problem revolves around the repetitive code updates that target the TB-specific hierarchy and test set. Consequently, a certain amount of ramp-up is necessary for each engineer that has to do adjustments. On the highest TB abstraction layer, the proper implementation approach should feature control knobs driven from outside the SV language.

The above approach comes with three advantages. Firstly, the format defining a testcase becomes human-readable. Secondly, one single compilation of the TB is able to define different test combinations. Thirdly, the test definition can be done as a text format, which allows the engineer to employ any other programming language or/and tools supported by the working environment. All these advantages are boosting our work efficiency.

III. ARCHITECTURE AND SCALABILITY

For a long time, the common understanding is that UVM has fixed the needs of TB architectures. I tend to disagree, UVM is putting a number of tools in the hands of the user but does not provide a clear guideline on how to use them. Each company defines a way of working, but most of the time, for the sake of steady advancement, that way of working has to accommodate the work and not the other way around.

To make an analogy, in the world of photography, having a good camera is necessary and you can find guidelines for each setting that camera has, and you can learn everything there is to learn about the parameters you can adjust. Having the knowledge is crucial, but it won't guarantee you will take good pictures. The composure of each shot is something that's entirely in the hand of the photographer, and each shot would need different settings in order to look better. The guidelines will never replace a good eye, so a guideline that cannot be bypassed might end up hurting in the long run.

However, most photographers use simplified modes on the camera when moving around shooting different composures and at different levels of light. They keep control on the knobs they deem as being important and let the camera figure out the others. I believe the current verification methodologies can benefit from the same mindset.

I need these VIPs, I need them to run these sequences, and I want to control configuration variables with this specific level of control. I don't need to care where the VIP is instantiated, I don't need to care how the sequence is created or started, as long as it runs at the right time. Also, I don't need to care how the control knobs are propagated in the environment, as long as they end up in the right place. So far, so good, but the needs change, the number of VIPs has to be scaled, or they have to be swapped, the sequences need to change to accommodate new/changed functionality and the constraints need to evolve. What I do care about is what needs to happen to make these changes. Based on the TB architecture, the changes can be small or big and can affect other functionality.

To conclude, I need the control over a series of parameters that enable or disable all the functionality of the TB. I don't need to care how the environment is built, what the hierarchy is, or how the control knobs are propagated. What I care about is how I add new components, how I add new control knobs and how I implement sequences that are independent to each other. The purpose of this paper is to provide a layer between UVM and the user's TB that does exactly what the simplified modes do on a camera. Specifically, it takes away control of implementation choices that do not bring verification value and increase the TB code complexity that can bolster the chances of inserting unintentional errors and bugs.

A. *Components, Objects and a Flat Hierarchy*

The first step in improving the ease of use, is to have an automated way of defining new objects and components in the environment or in any other component hierarchical levels like containers or sub-environments. Our approach provides a series of base classes built on top of UVM that dynamically facilitates the instantiation and creation of these components starting from text format definitions.

The key is to have a single point of insertion for all new functionality, keeping the test and the main environment static. This also offers a single anchoring point for engineers that have to ramp up on a new project. Having a predetermined structure makes solo exploration of the TB much simpler and makes the engineer less dependent on support from peers. The only requirement is a basic understanding of the principles explained in this paper.

B. *Stand alone sequence library*

Similar to the components, the sequences don’t exist per se. They are rather instantiated when needed through text formatted control knobs. This means that the virtual sequences have to be made modular and all randomizable variables should be propagated as control knobs using the functions that we provide.

An independent testcase can be defined as a collection of sequences run in a particular order with a particular set of constraints. The end goal of our architecture is to have completely modular and independent sequences that can be mixed together, either serially or parallel, as required by the necessary scenario. If this requirement is met, as well as the requirement of having all control knobs propagated through the predefined functions, then that means that any independent testcase, for which the functionality is available in the TB, can be created without having to touch the SV code again.

C. *One test to rule them all*

Furthermore, if all these sequences and all of their variables are available through text-based inputs, outside of the SystemVerilog environment, the concept of testcase becomes abstract, tied to an idea and composed of a collection of text inputs.

Creating new testcases becomes trivial, and the need for more files and more classes in order to create a new testcase, is removed. This is how the road from idea to implementation gets shorter and thanks to the newly defined format, it opens a new universe in terms of automation possibilities.

D. *External control*

A key idea in this architecture is the capability of controlling testbench entities from outside of it (external control). This idea transforms itself into an architecture requirement which defines a general recipe. This recipe is used to dynamically accommodate any kind of verification requirement.

Let’s have a look at an *uvm_component*. How is this component tied to the verification environment? Well, we can observe several tying points: a *parent component*, a *component type*, a *component name* and a *number of instances*. Having these inputs for each defined/definable component allows the environment to be scaled up and down indefinitely without touching the code, given a mechanism that facilitates the component creation.

Observing the other elements of a TB, e.g., the objects, would require a *parent name*, an *object type*, an *object name* together with the *controllable variables* and their *corresponding values* for this particular scenario.

Similar to the objects, the sequences would require a *sequence type*, a *sequence name*, a *parallelism status* (serial or parallel), the *controllable variables* and their *corresponding values* for this particular scenario.

IV. DECOUPLED STIMULI CONTROL

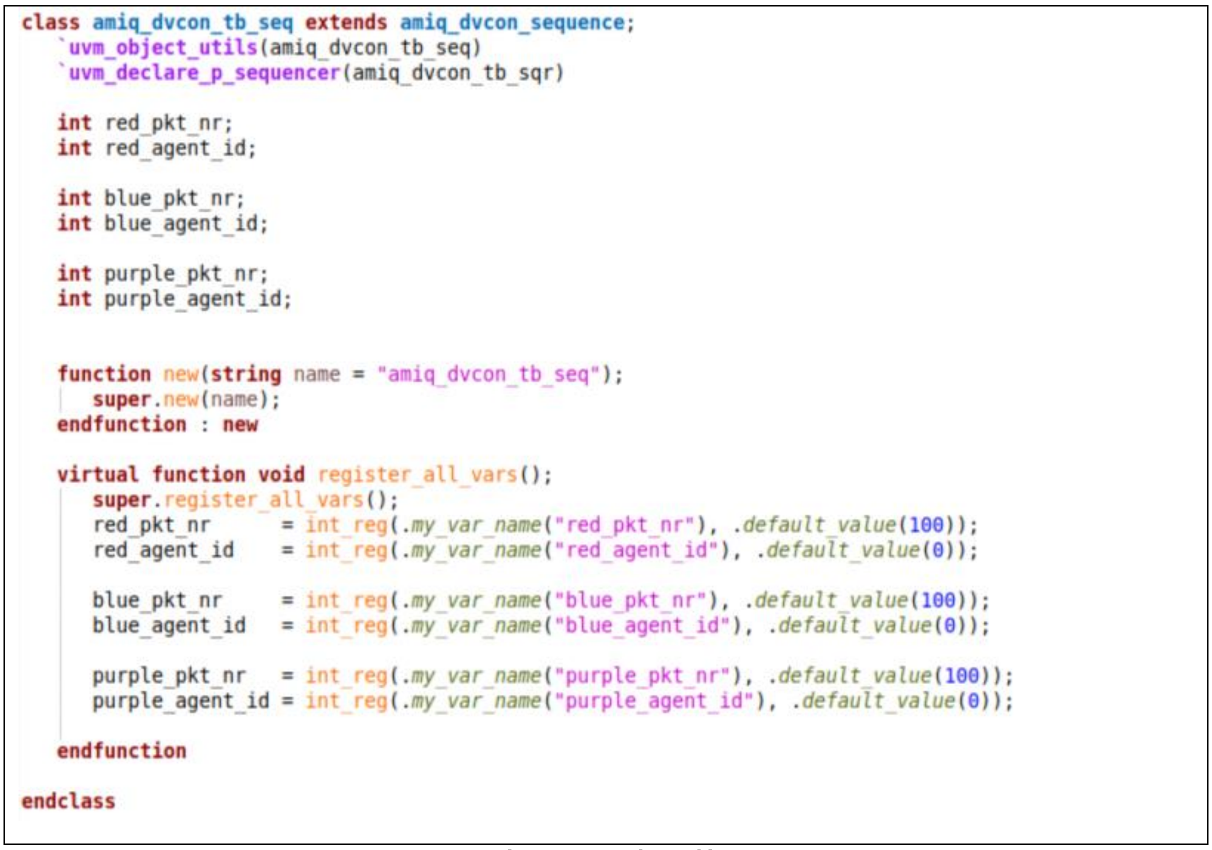
A. Control of stimuli at runtime without recompilation

The concept of sequence facilitates the flow of actions in a particular testcase. Stimuli, in an abstract form, are described as a collection of serial and parallel sequences. The stimulus pattern is controlled by flags that are either randomized or constrained in a virtual sequence. The randomization of the virtual sequences usually happens in the testcase. This approach is inefficient, as any change in the traffic pattern requires either writing a new sequence or manually modifying existing constraints. As we mentioned above, the testcase only exists as an idea and a collection of parameters. Having a method running outside of SystemVerilog (at run-time) and taking the control away from the SystemVerilog code, is a better approach in our opinion.

Our proposed solution takes advantage of the existing command line input functions and builds on top of it an infrastructure that would allow external control of any variable inside an object or sequence, with minimum changes to the existing code. We provide the base classes that encapsulate all the required functionality and enables control of any control knob using plusargs passed directly to the simulator without recompilation.

B. Randomization changes, adjustments and directed tests

Figure 1 contains a snippet of code for a simple sequence that can be run in our novel environment described in the previous section.

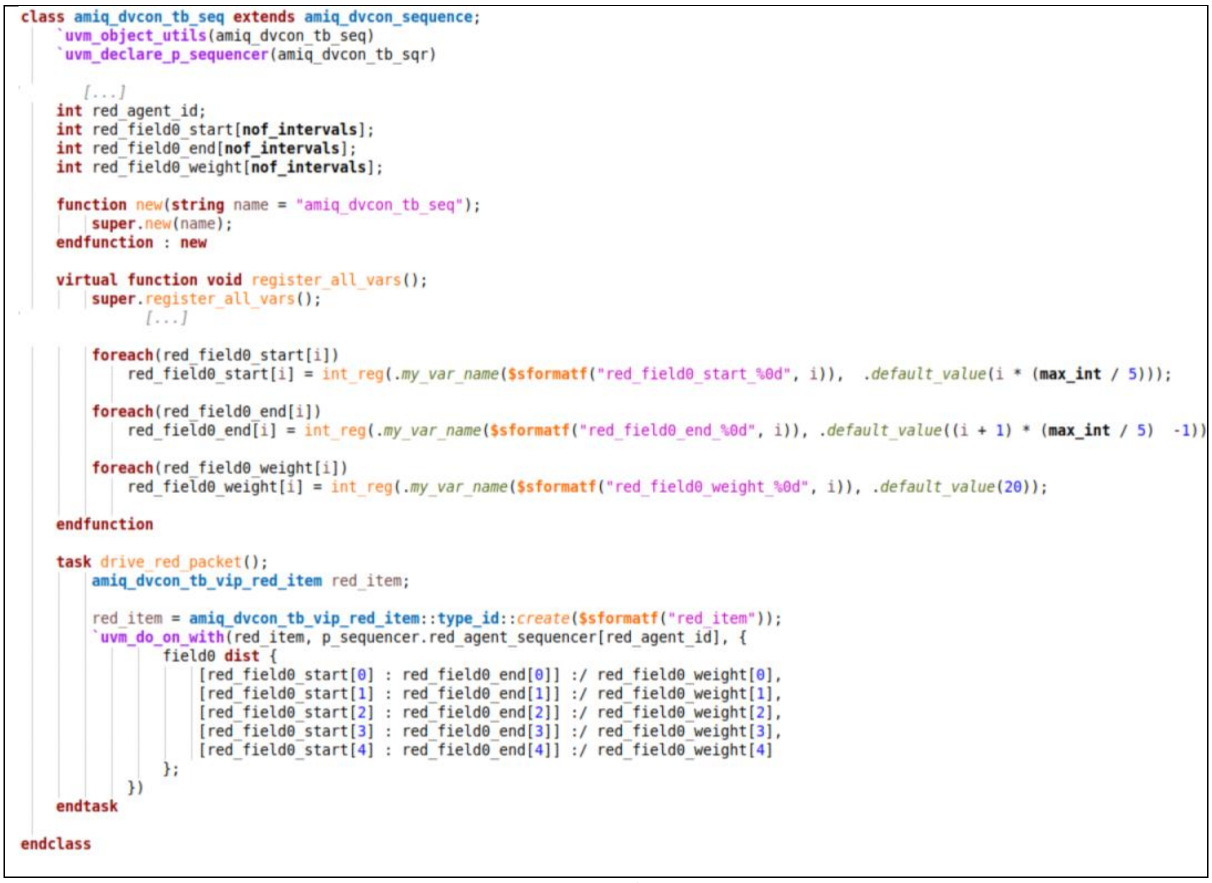


```
class amiq_dvcon_tb_seq extends amiq_dvcon_sequence;
   `uvm_object_utils(amiq_dvcon_tb_seq)
   `uvm_declare_p_sequencer(amiq_dvcon_tb_sqr)

   int red_pkt_nr;
   int red_agent_id;

   int blue_pkt_nr;
   int blue_agent_id;

   int purple_pkt_nr;
   int purple_agent_id;


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

   virtual function void register_all_vars();
      super.register_all_vars();
      red_pkt_nr      = int_reg(.my_var_name("red_pkt_nr"), .default_value(100));
      red_agent_id    = int_reg(.my_var_name("red_agent_id"), .default_value(0));

      blue_pkt_nr     = int_reg(.my_var_name("blue_pkt_nr"), .default_value(100));
      blue_agent_id   = int_reg(.my_var_name("blue_agent_id"), .default_value(0));

      purple_pkt_nr   = int_reg(.my_var_name("purple_pkt_nr"), .default_value(100));
      purple_agent_id = int_reg(.my_var_name("purple_agent_id"), .default_value(0));

   endfunction

endclass
```

Figure 1 Simple sequence with variable registration

After registering the variables, modifying the stimuli is trivial. Starting traffic on *my_red_agent_1* instead of *my_red_agent_0* is done by passing the following plusarg: ***+amiq_dvcon_tb_seq_red_agent_id=1***. To send 3000 blue packets instead of just 100, it is necessary to pass ***+amiq_dvcon_tb_seq_blue_pkt_nr=3000*** to the simulator.

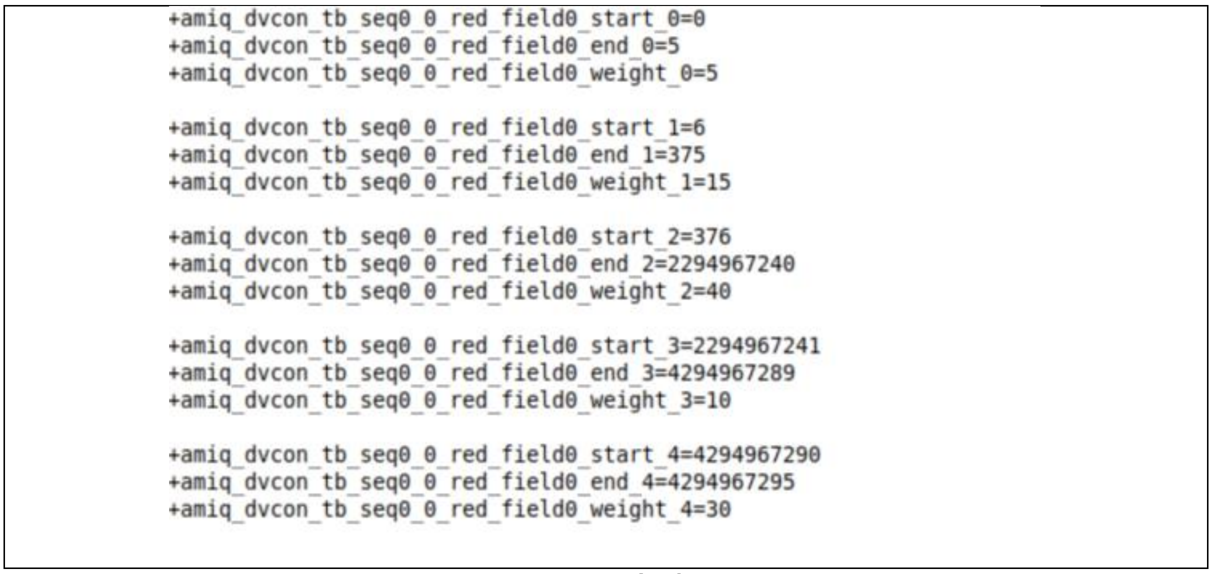
For more precise control of the randomization of each field, it is necessary to define constraints that are based on variables which can be updated via plusargs. In Figure 2 we show how to define a constraint block for a distribution, in which both the ranges and the weights can be controlled from the command line.



```
class amiq_dvcon_tb_seq extends amiq_dvcon_sequence;
    `uvm_object_utils(amiq_dvcon_tb_seq)
    `uvm_declare_p_sequencer(amiq_dvcon_tb_sqr)

        [...]
    int red_agent_id;
    int red_field0_start[nof_intervals];
    int red_field0_end[nof_intervals];
    int red_field0_weight[nof_intervals];

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

    virtual function void register_all_vars();
        super.register_all_vars();
                [...]

        foreach(red_field0_start[i])
            red_field0_start[i] = int_reg(.my_var_name($sformatf("red_field0_start_%0d", i)),  .default_value(i * (max_int / 5)));

        foreach(red_field0_end[i])
            red_field0_end[i] = int_reg(.my_var_name($sformatf("red_field0_end_%0d", i)), .default_value((i + 1) * (max_int / 5)  -1))

        foreach(red_field0_weight[i])
            red_field0_weight[i] = int_reg(.my_var_name($sformatf("red_field0_weight_%0d", i)), .default_value(20));

    endfunction

    task drive_red_packet();
        amiq_dvcon_tb_vip_red_item red_item;

        red_item = amiq_dvcon_tb_vip_red_item::type_id::create($sformatf("red_item"));
        `uvm_do_on_with(red_item, p_sequencer.red_agent_sequencer[red_agent_id], {
                field0 dist {
                    [red_field0_start[0] : red_field0_end[0]] :/ red_field0_weight[0],
                    [red_field0_start[1] : red_field0_end[1]] :/ red_field0_weight[1],
                    [red_field0_start[2] : red_field0_end[2]] :/ red_field0_weight[2],
                    [red_field0_start[3] : red_field0_end[3]] :/ red_field0_weight[3],
                    [red_field0_start[4] : red_field0_end[4]] :/ red_field0_weight[4]
                };
            })
    endtask

endclass
```

Figure 2 Range and weight control for a randomized distribution

In Figure 2, *field0* is randomized by default, using a normal distribution. To customize the randomization for a particular test, the arguments from Figure 3 are passed to the simulator.



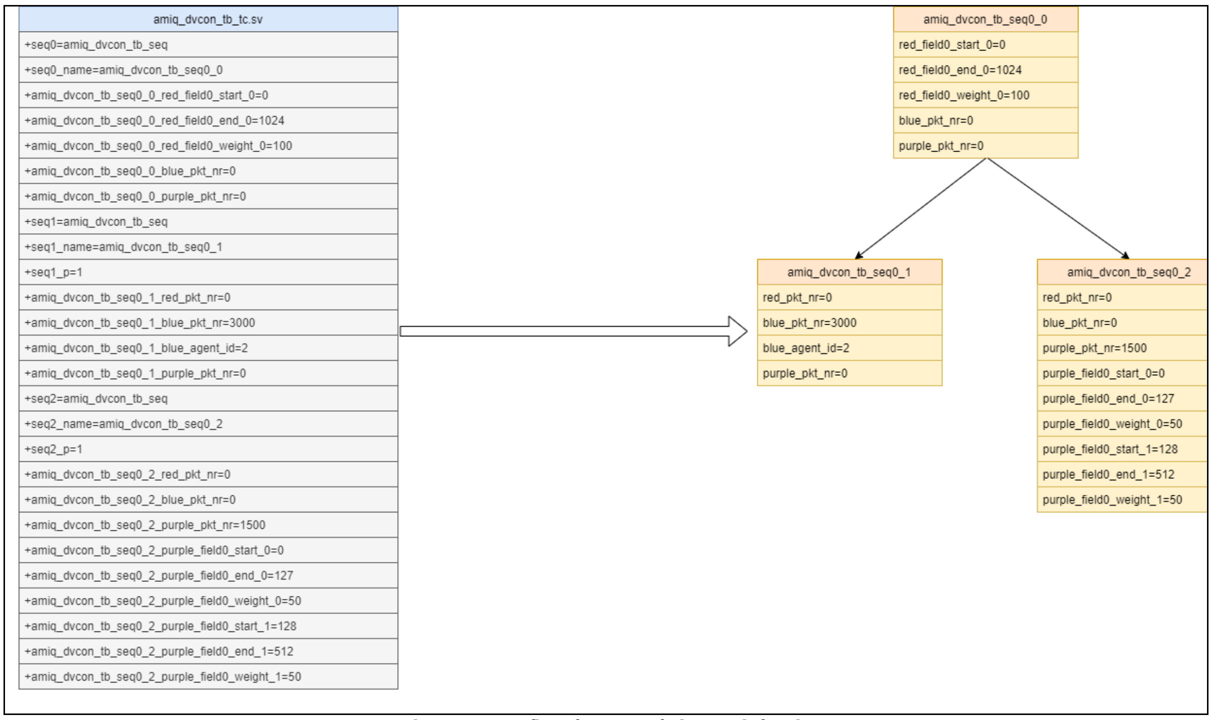
```
+amiq_dvcon_tb_seq0_0_red_field0_start_0=0
+amiq_dvcon_tb_seq0_0_red_field0_end_0=5
+amiq_dvcon_tb_seq0_0_red_field0_weight_0=5

+amiq_dvcon_tb_seq0_0_red_field0_start_1=6
+amiq_dvcon_tb_seq0_0_red_field0_end_1=375
+amiq_dvcon_tb_seq0_0_red_field0_weight_1=15

+amiq_dvcon_tb_seq0_0_red_field0_start_2=376
+amiq_dvcon_tb_seq0_0_red_field0_end_2=2294967240
+amiq_dvcon_tb_seq0_0_red_field0_weight_2=40

+amiq_dvcon_tb_seq0_0_red_field0_start_3=2294967241
+amiq_dvcon_tb_seq0_0_red_field0_end_3=4294967289
+amiq_dvcon_tb_seq0_0_red_field0_weight_3=10

+amiq_dvcon_tb_seq0_0_red_field0_start_4=4294967290
+amiq_dvcon_tb_seq0_0_red_field0_end_4=4294967295
+amiq_dvcon_tb_seq0_0_red_field0_weight_4=30
```

Figure 3. Custom distribution

V. Dynamic Execution Flow

Most verification environments have multiple tests, instantiating one or more virtual sequences that hardcodes the test flow. Any updates, however trivial, require touching the code and recompiling the TB.

The philosophy behind our approach highlights the lack of necessity to have any hardcoded elements in our test flow. The base test doesn't have any sequences instantiated, rather it looks for the plusarg **+seq#** to find the sequence types, creates them and runs them either in parallel or serially based on specific tags (**+seq#_p**). This way, the execution flow can be controlled completely from the command line or using a text file passed to the simulator. Figure 4 depicts the execution flow of a simple test, where the first sequence is run in serial, sending red packets, while the next two are run in parallel sending blue and purple packets.



```
amiq_dvcon_tb_tc.sv
+seq0=amiq_dvcon_tb_seq
+seq0_name=amiq_dvcon_tb_seq0_0
+amiq_dvcon_tb_seq0_0_red_field0_start_0=0
+amiq_dvcon_tb_seq0_0_red_field0_end_0=1024
+amiq_dvcon_tb_seq0_0_red_field0_weight_0=100
+amiq_dvcon_tb_seq0_0_blue_pkt_nr=0
+amiq_dvcon_tb_seq0_0_purple_pkt_nr=0
+seq1=amiq_dvcon_tb_seq
+seq1_name=amiq_dvcon_tb_seq0_1
+seq1_p=1
+amiq_dvcon_tb_seq0_1_red_pkt_nr=0
+amiq_dvcon_tb_seq0_1_blue_pkt_nr=3000
+amiq_dvcon_tb_seq0_1_blue_agent_id=2
+amiq_dvcon_tb_seq0_1_purple_pkt_nr=0
+seq2=amiq_dvcon_tb_seq
+seq2_name=amiq_dvcon_tb_seq0_2
+seq2_p=1
+amiq_dvcon_tb_seq0_2_red_pkt_nr=0
+amiq_dvcon_tb_seq0_2_blue_pkt_nr=0
+amiq_dvcon_tb_seq0_2_purple_pkt_nr=1500
+amiq_dvcon_tb_seq0_2_purple_field0_start_0=0
+amiq_dvcon_tb_seq0_2_purple_field0_end_0=127
+amiq_dvcon_tb_seq0_2_purple_field0_weight_0=50
+amiq_dvcon_tb_seq0_2_purple_field0_start_1=128
+amiq_dvcon_tb_seq0_2_purple_field0_end_1=512
+amiq_dvcon_tb_seq0_2_purple_field0_weight_1=50
```

```
amiq_dvcon_tb_seq0_0
red_field0_start_0=0
red_field0_end_0=1024
red_field0_weight_0=100
blue_pkt_nr=0
purple_pkt_nr=0
```

```
amiq_dvcon_tb_seq0_1
red_pkt_nr=0
blue_pkt_nr=3000
blue_agent_id=2
purple_pkt_nr=0
```

```
amiq_dvcon_tb_seq0_2
red_pkt_nr=0
blue_pkt_nr=0
purple_pkt_nr=1500
purple_field0_start_0=0
purple_field0_end_0=127
purple_field0_weight_0=50
purple_field0_start_1=128
purple_field0_end_1=512
purple_field0_weight_1=50
```

Figure 4. The execution flow for a set of plusarg defined sequences

One can observe that any number of sequences can be defined, giving only their type, name, parallelism status and the constraints for the relevant fields. Any field that doesn't have a defined plusarg would use the default value given in the sequence definition of the "**register_all_vars()**" function.

The next opportunistic step is to automate this process for a much easier way of defining plusargs and also a better way to keep track of them especially for debugging purposes.

VI. Automation

A. *Creating your own software for the TB*

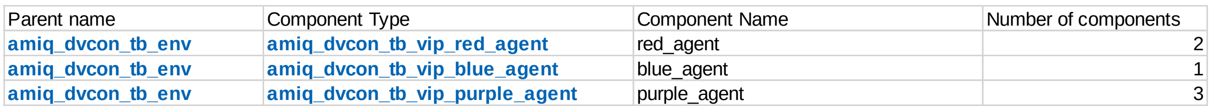
Since the controls are taken out of the HDL language, any number of tricks and scripts can be employed to automate the process of generating, modifying, and visualizing the environment, the sequence hierarchy and the variable fields.

To give an example, we developed a simple script that allows the definition of environments and testcases based on excel spreadsheets. The requirement is to use a particular header for each excel spreadsheet based on what type of plusarg collection you want to generate.

The generation possibilities are components, objects and sequences which from our research are all the building blocks you need for creating any TB. In the following chapters we show examples on a simple use case, the one that is also present in the self-TB of our library. The goal is to successfully generate and scale any kind of environment given a collection of VIPs, scoreboards, coverage collectors and miscellaneous objects.

B. *Environment control*

The environment is per se a container for all the components and objects existing in the TB. As such, in our methodology, any number of components and objects can be instantiated automatically at run-time using plusargs. For this purpose, an excel spreadsheet similar to the one in Figure 5 has to be created.



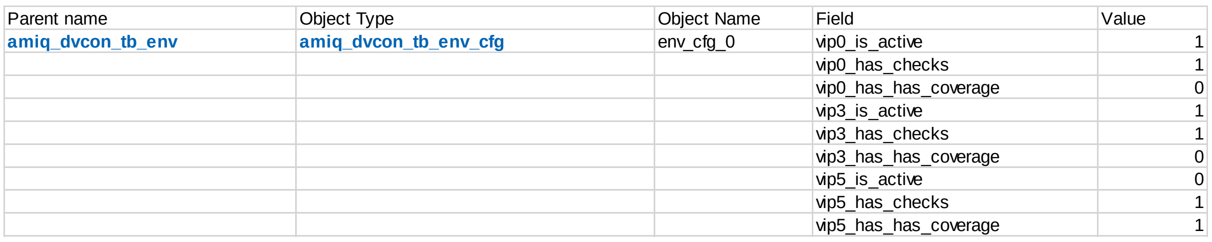
Parent name	Component Type	Component Name	Number of components
amiq_dvcon_tb_env	amiq_dvcon_tb_vip_red_agent	red_agent	2
amiq_dvcon_tb_env	amiq_dvcon_tb_vip_blue_agent	blue_agent	1
amiq_dvcon_tb_env	amiq_dvcon_tb_vip_purple_agent	purple_agent	3

Figure 5. Automated Environment Creation

The script will take this as an input and create a text file with the name of the spreadsheet and containing the formatted plusargs for all the lines in the excel. After the text file is fetched by the simulator, the results point out two red agents, one blue and three purple agents will be created with the corresponding name "***color*_agent_****nr******" under the component "**amiq_dvcon_tb_env**", no matter where this component is instantiated and no matter if it is a static defined component in the TB or it is created using this component generation mechanic. Thus, any hierarchy, of any depth, can be created with our novel system.

Any number of spreadsheets can be created in an excel document and all of them will generate one text file that can serve as an input. This is important because some parts of the TB like the sequences might be dynamic, and you would continuously update them from one regression to the next one while the environment might be static for the most part. Multiple excel files can be created containing different spreadsheets based on the frequency of their usage. The script accepts any number of excel files at a time and each of them contains any number of spreadsheets of any header type.

For generating configuration objects or any other miscellaneous objects the same concept is valid, with a different header on the first row of the excel spreadsheet



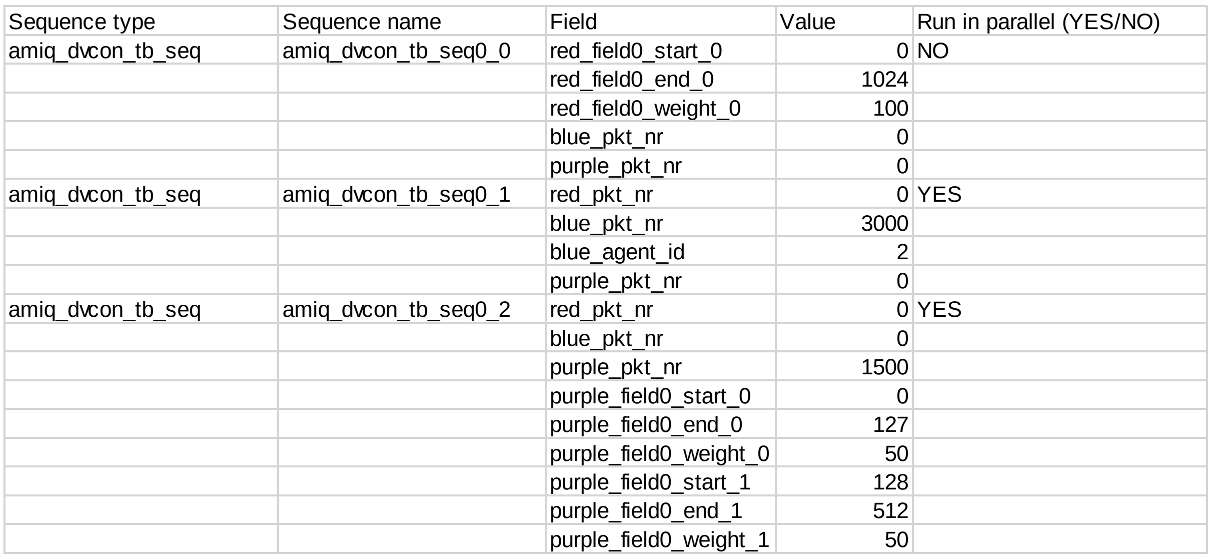
Parent name	Object Type	Object Name	Field	Value
amiq_dvcon_tb_env	**amiq_dvcon_tb_env_cfg**	env_cfg_0	vip0_is_active	1
			vip0_has_checks	1
			vip0_has_has_coverage	0
			vip3_is_active	1
			vip3_has_checks	1
			vip3_has_has_coverage	0
			vip5_is_active	0
			vip5_has_checks	1
			vip5_has_has_coverage	1

Figure 6. Object creation

Observe in Figure 6 that one instance of that object type with name “env_cfg_0” has multiple fields that are set to certain values. In plusarg format, the object creation has a similar format to the component with a keyword difference and each individual field uses the name. Through the script usage, this format is simplified and becomes human readable.

C. *Scenario creation*

The last piece of the puzzle is creating the list of target scenarios. Given the fact that the testcase is always the same and that it is based on the base testcase in our library, scenarios can be defined as a collection of sequences with a distinct set of constraints. To define a scenario via an excel spreadsheet the following format should be considered:



Sequence type	Sequence name	Field	Value	Run in parallel (YES/NO)
amiq_dvcon_tb_seq	amiq_dvcon_tb_seq0_0	red_field0_start_0	0	NO
		red_field0_end_0	1024	
		red_field0_weight_0	100	
		blue_pkt_nr	0	
		purple_pkt_nr	0	
amiq_dvcon_tb_seq	amiq_dvcon_tb_seq0_1	red_pkt_nr	0	YES
		blue_pkt_nr	3000	
		blue_agent_id	2	
		purple_pkt_nr	0	
amiq_dvcon_tb_seq	amiq_dvcon_tb_seq0_2	red_pkt_nr	0	YES
		blue_pkt_nr	0	
		purple_pkt_nr	1500	
		purple_field0_start_0	0	
		purple_field0_end_0	127	
		purple_field0_weight_0	50	
		purple_field0_start_1	128	
		purple_field0_end_1	512	
		purple_field0_weight_1	50	

Figure 7. Scenario generation

Similar to the objects, a type name and all of the fields are set. The difference is that a parent name is not required since this exists under the “+UVM_TESTNAME” test. Also, to allow the creation of sequence trees, the serial/parallel flag has to be provided. Any number of spreadsheets grouped in any number of excel files can co-exist to establish a comprehensive library of testcases for any verification project. This format can also be reviewed by non-technical people or by the software team and used for documentation.

The choice of defining plusargs as the prime mean of control instead of using hardcoded sequence instantiations and constraint blocks, brings forth many other options of organizing the verification documents, planning and trouble reports.

VII. Conclusions And Future Work

A. *Conclusions*

UVM/SystemVerilog is far from being insufficient for the current complexity of IC projects. The problem consists in its many degrees of freedom that permit too much overhead and variation.

We aim at providing examples and coding patterns that adhere to the guidelines presented above, as well as a fully working TB and an example of a framework for TB generation.

The ideas presented in this paper are the result of years of experiments and learning from mistakes on testbenches that have been developed by many engineers, each with a different background and vision.

The title summarizes the issue that brought forward the need for a development of such a TB: **the creativity of each individual is undermining the collective effort of the entire team**. The solution with which we experimented introduces a pragmatic architecture that has a predefined mechanism for adding and removing components (scoreboards, VIPs, etc.). Similarly, the scenarios can be created by mixing modular sequences that are created and scheduled at run time, instead of being hardwired instances in a test.

Basically, we take a step back from the UVM dogma and capitalize on good ol' OOP principles.

B. Future work

The automation chapter and the script we created for both the TB and the scenario generation is just a small piece of insight in the kind of potential that this underlying approach has. On our roadmap, we have some additional scripts and software that establish a feedback loop between the results of the regressions/simulations and the plusarg inputs that generate scenarios. This is a good step in the direction of establishing self-improving testbenches.

C. Library availability

Note to reviewer: The draft version for the final paper doesn't have a link available for downloading the open-source library. This will be provided in a later upload before the final version deadline. The reason is a pending/ongoing review on our solution's features, and we are doing this on our own time. We are also aware that the size goes beyond the requirement and that we are missing references and acronyms. This is a draft pending adjustment.