2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Portable Stimulus Driven SystemVerilog/UVM verification environment for the verification of a high-capacity Ethernet communication endpoint

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

Ionut Tolea, Amiq Consulting, Bucharest, Romania (*ionut.tolea@amiq.com*)

***Abstract*—The scope of this paper is to present the steps taken and the challenges faced when using Portable Stimulus(PSS) as an abstraction layer on top of a SystemVerilog/UVM verification environment. The goal is the verification of a highly configurable, high-speed, communication endpoint, covering complex network scenarios and system-level corner cases. PSS is used in conjunction with SystemVerilog/UVM to increase verification efficiency by avoiding "scenario flooding" and keep a tight control of the verification space.**

All stimuli are defined in the test-bench and they are used to construct directed/random scenarios by utilizing a PSS generation model. The flow of the project in this case requires the SV/UVM VE to keep up with the guidelines for reusability while having an architecture that is compliant with the PSS mechanics of scenario generation. The paper addresses the possible issues and good practices discovered while implementing this verification strategy.

Keywords— Hardware Verification; Portable Stimulus; SystemVerilog; UVM

I. INTRODUCTION

The present paper addresses the challenges and proposes solutions for system-level verification of complex devices in the era of IoT. The scope of verification at system level, especially when dealing with a network of devices is not fully visible from the DUT's point of view. The verification scenarios are composed of software procedures that have a given behavior and a certain set of side effects when translated to hardware stimuli. By using PSS, we aim to bring these software procedures into a more physical form and break them in steps that make sense from both the hardware and software point of view.

Today, the bridge between a system architect's defined scenario and a testcase implementation is solely verification engineer's understanding of the system, including the software which might not exist at the time of the DUT verification. The validation of the implementation is also close to impossible if the architect is not familiar with verification or low-level hardware design. PSS gives an additional abstraction layer which fills the gap between the idea and the implementation, while also providing visual representation of the scenarios.

The current method of using multiple layers of sequences to organize actions have limitations in terms of scalability and the purpose of these higher layer sequences can't always be strictly controlled. Timing issues can also become problematic. Small changes of the scenario can translate into massive amounts of coding.

By using PSS, the role of the hardware verifier will be to create a test-bench and an adaptation layer that works with the complete set of software procedures defined as actions. This way, we target to gain a better way of verifying system-level scenarios by creating testcases composed of software actions rather than bus transactions.

II. SYSTEM LEVEL AND NETWORK SCENARIOS

A. Functional verification on sub-system level

The main problem when doing verification on sub-system is that even though the intent can be well defined, the collection of actions that form real life scenario can be quite different from what stimulus should be applied to the DUT to get the desired result. When planning, the verification strategy will often focus on this collection of actions, rather than the timing and stimuli used on the RTL.

For example, the initial configuration of the DUT in the real system will be composed of transactions on a low speed serial interface connected to an onboard CPU as well as transactions on a high-speed parallel bus connected from a remote client. The action of configuring specific blocks in the sub-system has two general properties: the values that are deposited in the registers and the interface that transports the configuration accesses. The software and hardware scenarios are identical on a logical level but have different translations in physical stimuli. This translates into a different adaptation layer for different scopes of verification.

Another advantage of using action-level granularity for the steps taken in a scenario is that the reusability of an environment is maximized. By having a complete collection of actions that a system can support, it is possible to construct a test-bench infrastructure that will be able to accommodate all the foreseeable scenarios that have to be tested. On sub-system verification the most time-consuming task is modifying existing components or sequences to accomplish new scenarios, pushing UVM to its limits. With an increasing number of directed testcases the flexibility that we can afford in terms of code modifications decreases at the same time with code maintainability.

To avoid the problem of modifying existing layers of sequences to accomplish new scenario goals, the most used method is to stick to one virtual sequence per testcase which chains physical sequences and takes care of the synchronization issues. However, this greatly increases the amount of code necessary to accomplish the verification goals and it also brings higher debug times than development times due to lack of fine tuning and comprehensive control between the intended behavior and the hardware stimuli.

B. PSS as an abstraction layer

Using PSS as an additional layer in verification imposes a better control for scenario definition and through its nature it imposes a better planning for the verification strategy. The existence of specific actions that the system can undergo allows for the test-bench to be tailored alongside these actions. We also expect to improve the focus factor for the verification environment development and to significantly decrease the debug time.

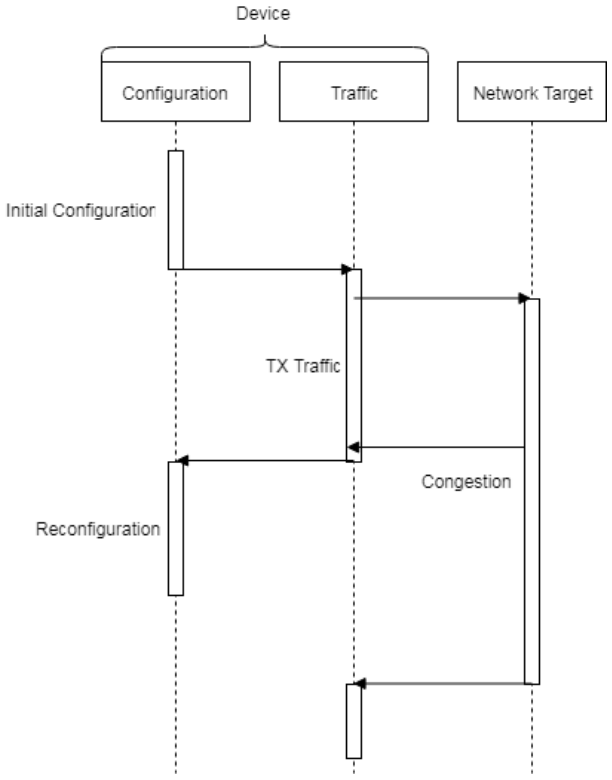

Figure 1. Flow control scenario

To reap the benefits of this approach a series of guidelines need to be followed for the test-bench:

- All actions should be defined as stand-alone sequences that have no physical ties to other actions

- All test-bench configuration that needs to happen at runtime for specific actions should be defined as objects of the action in the PSS model
- If an action at system level is composed of multiple modular sequences that require to be used in a specific order, a new sequence should be created which calls the modular sequences and configures them to the allowed extent. The configuration that the higher layer sequence does should be defined as objects of the action.
- Synchronization and timing can be defined as actions if it makes sense from the system point of view and it should translate to event triggers or delays in the test-bench
- The coverage and logical constraints for the actions/scenarios should be done in the PSS layer, while the protocol constraints for available interfaces should be done in the SV/UVM layer. Both gen-time and run-time PSS coverage should be activated to ease debug and widen the safety net.

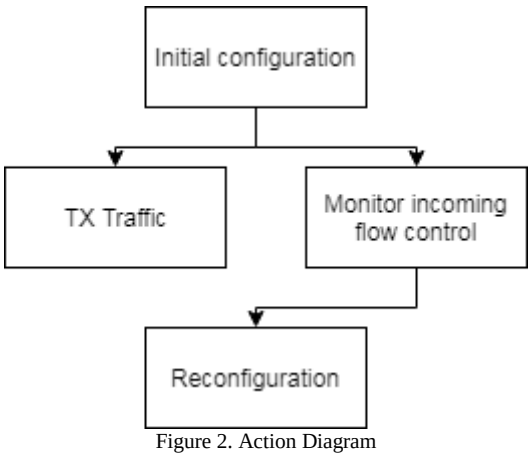

Figure 2. Action Diagram

In Figure 1, a simple flow control scenario is defined for an Ethernet endpoint. An initial configuration of the DUT needs to take place before the traffic can be initiated in the TX direction. We consider the scenario in which the target cannot handle the flow at the same line rate as the initiator, so the speed needs to be reduced. For this to happen, the target should send flow control frames to the initiator. The actions that need to take place are displayed in Figure 2. It should be noted that apart from the configuration and traffic actions which have direct physical interpretation to the DUT, the monitoring of flow control is a logical action that is translated to an event in the test-bench.

The general flow is that the monitoring should happen in parallel with the traffic, after the initial configuration, while the reconfiguration should happen serially after the monitoring action is finished (i.e. the event was triggered). By defining objects for the actions, the behavior of the DUT can be constrained from the PSS layer. For example, if the configuration can be done on multiple interfaces, that should be a randomizable resource of the action.

III. PSS IMPLEMENTATION

The purpose behind the usage of PSS is to provide generalization of concepts and the possibility of building complex scenarios based on the most basic transactions available in the TB. Each UVC available is most of the time a proven piece of software capable of providing error proof basic scenarios.

The problems most of the time arise when the complexity of the scenarios increase so much, that basic functionality doesn't accommodate the needs anymore and custom solutions, which most of the time diverge from the usual UVM guidelines, need to be implemented. The best approach has always been to create as many layers as possible with each increase in complexity.

By isolating the working layers on each iteration, it limits the bug sensible zone to just the uppermost layer. Unfortunately, this approach is limited and after a certain threshold, the implementation cannot be bug controlled efficiently. Bluntly said, the amount of debugging time spent on a project related to the development time increases exponentially with the complexity of the scenarios.

For the following code blocks in this paper, a DSL will be used for which we had most support as far as documentation and tools go, but the underlying concepts are valid for any kind of PSS implementation.

A. From sequences to actions

To simplify the TB, the best course of action would be to isolate the most complex logic and move it to the PSS abstraction layer. The result would be that the remaining sequences in the testbench would be able to accommodate the collection of actions defined in the PSS model.

Let's consider a simple DUT, composed of a MAC, a DMA, a filtering block and a control block connected to a CPU. We also consider that the DUT is configurable via three interfaces, a serial and a parallel one available in the system and a high-speed interface from external clients. We also consider that the MAC can receive and transmit data at different speeds.

Following the guidelines proposed in chapter II we create a series of actions that our DUT can undergo. The first thing that comes to mind is the possibility of configuring different blocks in the sub-system in different order and through different interfaces. The traffic might also be restricted in certain ways by what has been configured and what not.

The first step would be to create a container that would host the first action:

```
<'

Type line_rate_e {l_10G, l_25G, l_100G};
Type cfg_if_e {SERIAL, PARALLEL, HS};

component action_container {
        action mac_config {
                line_rate: line_rate_e;
                cfg_interface: cfg_interface_e;
                enable_rx: bool;
                enable_tx: bool;
                enable_tx_fc: bool;
                enable_rx_fc: bool;
                strip_header: bool;
                check_crc: bool;
                pause_watermark_high: bool;
                pause_watermark_low: bool;

                exec body SV #:
                mac_config_seq_type mac_cfg;
                `uvm_do_with (mac_cfg, {
                line_rate==<(<line_rate)>;
                cfg_interface<(cfg_interface)>
                …});
                end #;

        };
};

'>
```

Figure 3. Container component with a fully defined action

We can see that the arguments defined do not have a physical translation in the TB but are logical steps taken when creating a scenario. The extent of the objects inside an action would be the full capability of that certain functionality translated into knobs available as randomizable fields inside a sequence.

The advantage with this approach is that a series of tools are available which can help with the mix and randomization of the knobs allowing for a large selection of scenarios to be reached with the push of a button.

The scenarios can either be generated automatically towards the fill of a coverage model or manually defined using the collection of actions as building blocks.

Let us consider a fully defined collection of actions that the DUT can undergo:

```
<'

…

component action_container {
        action mac_config {
                …
        };
        action mac_config {
                …
        };
        action filter_config {
                …
        };
        action control_config {
                …
        };
        action send_rx {
                …
        };
        action send_tx {
                …
        };
};

'>
```

Figure 4. UVM sequence collection translated to PSS model actions

The next logical step would be to generate a scenario based on the defined actions. Most of the scenarios on sub-system scope are directed testcases aimed to fulfill a real-life use case. In this case we aim to randomize which blocks we configure and then to send interesting traffic based on the configuration done.

In the above code sequence, we extend the existing action library by adding a new layer of sequences. In the

```
<'
extend action_container {
        action minimum_setup {
                compound {
                        >sequence {
                                >dma_config;
                                >mac_config;
                                };
                        };
                };

        action optional_setup {
                configure_filter: bool;
                configure_control: bool;

                compound {
                        >a: if (configure_filter) {
                        >then: filter_config;
                        };
                        >b: if (configure_control) {
                                >then: control_config;
                        };
                };
        };
};
'>
```

Figure 5. Scenario action which encapsulates the parts
of the configuration phase and adds control knobs

first action defined, we consider that the DMA must be active before the MAC goes active, so we create a sequential execution of the two actions. Secondly, we have the optional configuration which consists of blocks that add

functionality to the sub-system but are not mandatory for basic use. Because they are not mandatory, two flags are added to the action which are randomized when a scenario is solved and which control if the configuration for a certain block takes place.

Like UVM sequence layering, the goal is to create a series of bigger actions that build upon the smaller ones. The rule followed here is that the translation to the UVM code is contained to the most basic actions. The reason is that the simplicity of the actions will in turn limit the debug necessary on the SV/UVM layer, which is most time-consuming. Using a PSS model is efficient only if the compartmentalization is respected.

Another concept used in UVM which can be translated to the PSS model is the top-to-bottom parameter propagation, or even more so, the dependency handling between different sequences. In SV/UVM, the bugs arise out of corner cases which are exposed by many iterations and a lengthy debug process. The visual representation available for the PSS model can help expose bugs even before running a testcase.

```
action traffic {
        total_rx_count: int [1..10000];
        constraint total_rx_count >= 1;
        total_tx_count: int[1..10];

        configure_control: bool;
        configure_filter: bool;
        compound {
                >p1: parallel {
                        >s1: sequence {
                                >* replicate {
                                        .count == total_rx_count;
                                        > send_rx {
                                                .invalid_config == configure_filter;
                                        };
                                };
                        };
                        >s2: sequence {
                                >* r: replicate {
                                        .count == total_tx_count;
                                        >i: if (configure_control) {
                                                >tx: send_tx;
                                        };
                                };
                        };
                };
        };
};
```

Figure 6 Traffic action and control knob propagation

In the above action, the objects refer to the control knobs of the configuration sequence. The way the sequence is written allows for the control knobs to be propagated from a testcase action. Allowing the same randomized object to control both the execution of a configuration action as well as constraining the content of the traffic proves the possibility of satisfying the dependencies between different UVM sequences.

The requirement to satisfy any scenario needs is to first define all the necessary characteristics of the basic actions, the same as they would be defined in a UVM sequence, as well as all the necessary dependencies between them. Layering and compartmentalization are tools through which this can be achieved with the shortest amount of debug and the least number of side effects.

The last and highest layer of the PSS model is the action which defines the testcase. The PSS model allows for any kind of actions to be chained together, so the extension of an existing testcase should always be possible without any code modifications in UVM/SV.

In the below code sequence, a random testcase creation is targeted which will use the previous defined actions. The concept of knob propagation is most visible here, the knobs are defined and constrained in this action and then passed down to the lower layer actions.

```
action test_case {
        max_rx_count:int;
        constraint max_rx_count in [1..10];
        configure_control: bool;
        configure_filter: bool;

        compound {
                >sequence {
                        >minimum_setup;
                        >optional_setup {
                                .configure_control == configure_control;
                                .configure_filter == configure_filter;
                        };

                        >traffic {
                                .total_rx_count == max_rx_count;
                                .configure_control == configure_control;
                                .configure_filter == configure_filter;
                        };
                };
        };
};
```

Figure 7 Action that defines a testcase

The above testcase can be solved multiple times and will result in different testcases, based on what configuration will happen. For example, if "configure_filter" will be randomized to 1, the filtering block will be configured and that would also trigger the "rx_traffic" to include illegal packets.

Also, the existence of "tx_traffic" is dependent on the configuration of the control block, so that will only take place if configure_control has been randomized to 1. The propagation and usage of the "configure_control" and "configure_filter" knobs can be followed in the actions defined earlier.

The tools we had available were able to generate full testcase files, ready to be included in our testcase suite package. While the sequence calls were a small part of the code, the tool also gave us the capacity of recreating different scenarios with one compilation, dramatically reducing the time it takes to run the entire suite.

IV. REAL LIFE PROJECT APPLICATION

To assess the usefulness of a PSS model, we created, following the rules mentioned in this paper, the actions necessary to build the directed testcases suite of a real project I have worked on. The result was that we were able to recreate the entire testcase suite by using randomized control knobs and coverage definitions in just a couple of days compared to the months it took when using only UVM/SV. The problem with using only UVM/SV is that the scope is sometimes lost in the implementation which lengthens the time necessary.

We also faced problems in the beginning because the testcases were not generated correctly, but due to the diagrams provided by the tool we were using, that debugging never got to SV, we solved any issue we encountered from the PSS model.

The challenge when verifying system and sub-system scenarios is to bridge ideas to available SV/UVM concepts. The definition of PSS actions grants a clear roadmap, a set of tasks that would enable the realization of the verification plan. The potential to be able to estimate the entire implementation based on the verification needs of a system or sub-system is a meaningful change. From past experiences, the focus factor plays a huge role into the efficiency of a project development. Knowing exactly what is needed from start to finish can accelerate the verification if the initial plan is correct.

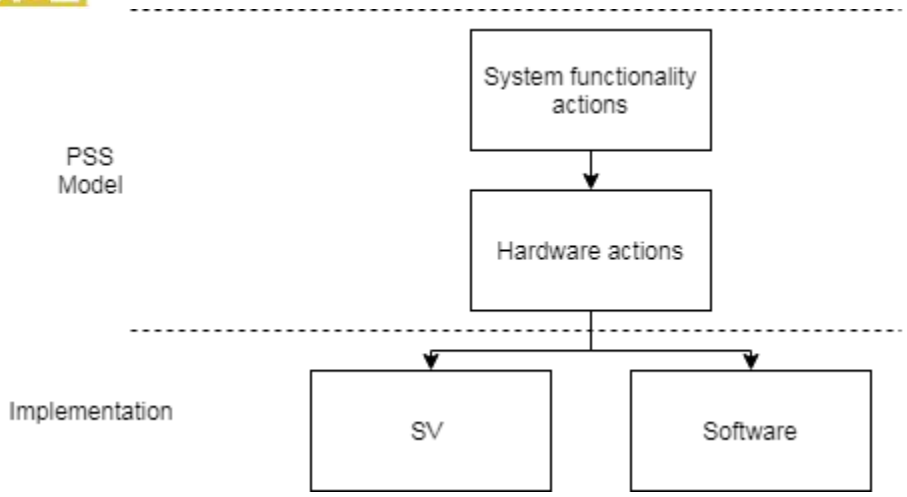

Figure 8 PSS Model layering for multiple platforms

The big question we had when doing the implementation was how suitable the PSS model will be for usage with other platforms, if we create it with just the RTL verification in mind. The answer is that we did not see any limitations. The only problem that can arise is that when doing transaction level verification, some of the system-level defined actions might translate to a collection of transactions on multiple interfaces. When working with UVM/SV, that would imply the existence of multiple sequences across multiple agents, so in some cases, the system-level action can be better represented for RTL verification as a collection of smaller actions.

Given the fact that the PSS model is much smaller than the UVM/SV environment needed for verification, the time investment necessary for a translation layer from the system layer actions to the transaction layer actions is negligible. To assure the portability of the model, the system level requirements should always be the base for the model, as UVM/SV will always be easier to adapt.

There may be actions that are solely applicable to functional verification on RTL, but verification of any digital device will always consist of configuration and traffic. The actions that the DUT can undergo, are sub-sets of those two, so the main thing that is different between using the PSS model with SV or with any other platform is the implementation that derives from those actions.

V. PRELIMINARY CONCLUSIONS

In conclusion, the PSS standard has all the necessary features to accommodate a higher abstraction layer over a general SystemVerilog/UVM test-bench. It also improves on areas like re-usability and it offers a tighter control over the test-bench features, which in term should decrease both development and debug times.

Due to the extensive support from the vendors for PSS tools, this approach will also bridge the communication gap between the verification engineers and the system architects by offering a common language through actions and scenario diagrams.

The biggest advantage is that a general recipe can be defined for test-bench development, in which actions should translate to actual code, while the testcases will be automatically generated through PSS tools. This will result in shorter time to market and better time estimation of the verification effort.

REFERENCES

[1] Accellera. “Portable Test and Stimulus Standard”

[2] Cadence. “Perspec System Verifier User guide”

[3] ANSI/IEEE 1800-2012 – IEEE Standard for SystemVerilog—Unified Hardware Design Specification and Verification Language

[4] IEEE 1800.2-2017 – IEEE Standard for Universal Verification Methodology Language Reference Manual