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

Andrei Vintila
AMIQ Consulting
9 Vespasian St.
Bucharest, RO 010921

Ionut Tolea
AMIQ Consulting
9 Vespasian St.
Bucharest, RO 010921

***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 bridge, 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.

I. INTRODUCTION

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 SW procedures that have a specific behavior and a set of side effects when translated to HW stimuli. By using PSS, we aim to translate the SW procedures into a more physical form and break them in steps that make sense from both the HW and SW 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 SW which might not exist at the time of the DUT verification. The validation of the implementation is close to impossible if the architect is not familiar with verification or low-level HW design. PSS provides an 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 cannot always be strictly controlled. Timing issues can also become problematic. Slight changes of the scenario can translate into massive amounts of coding.

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

Using action-level granularity for the steps taken in a scenario maximizes the reusability of an environment. Using PSS together with a complete collection of actions supported by the DUT, reduces the number of directed testcases and increases the verification environment flexibility and the code maintainability.

B. *PSS as an abstraction layer*

Using PSS as an additional layer in verification helps with the verification planning by keeping the focus on relevant actions and their relevant mixtures (i.e. scenarios). In turn this approach will improve the focus factor for the verification environment development and significantly decrease the debug effort.

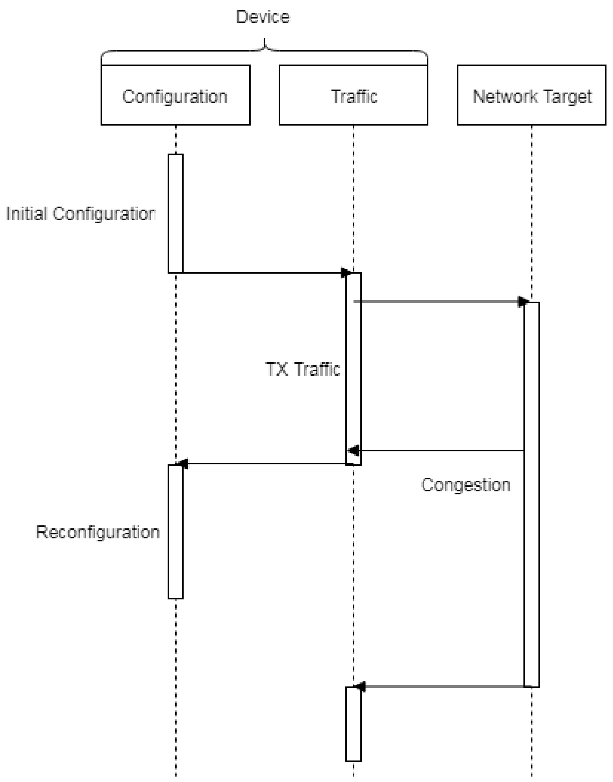


Figure 1. Flow control scenario

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

- Actions are defined as stand-alone sequences that have no physical relationship 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.

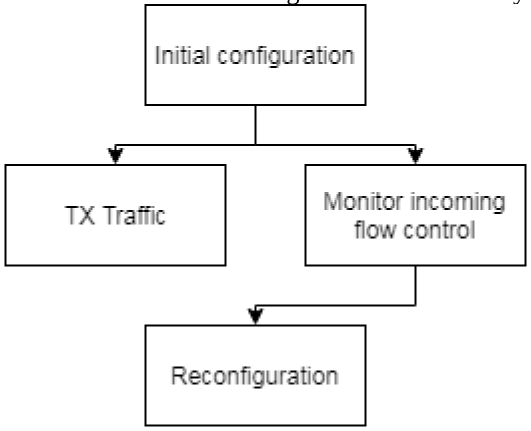


Figure 2. Action Diagram

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.

Usually the problems arise when the complexity of the scenarios increases, that basic functionality does not 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, PSS 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 us 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.

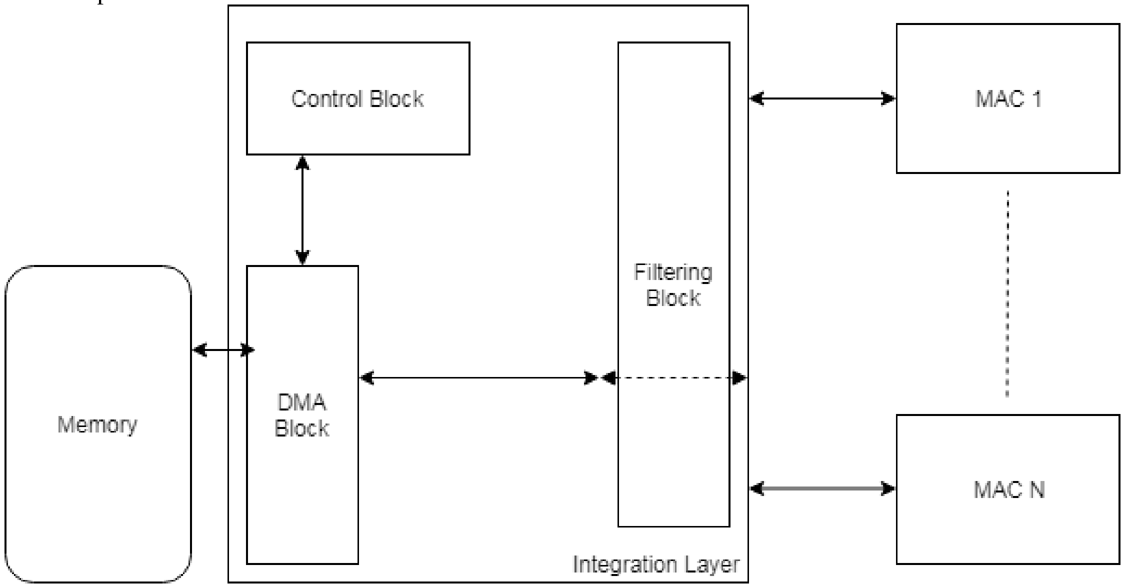


Figure 3. System Diagram

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:

```
// Configuration for the DMA block ; Necessary for any kind of traffic ; Controls accesses to memory
action dma_config {
    // Used to dictate the maximum buffer size for received packets written into memory ;
    // Packets above the threshold will be dropped
    rand int in [1..10000] max_size_buffer;

    // Used to dictate if the data will be written as soon as received or only when a full packet is r
    // Enable only if ethernet header is passed through and if the len_type is used as length
    // Maximum buffer size is the maximum ethernet packet size for length_defined packets
    rand bool write_through;

    // Physical interface on which this action will be executed
    rand cfg_interface_e interface_used;

    // Once the configuration is done we set the state to done
    output dma_state_s dma_state;
    constraint dma_state.config_done == true;

    exec body SV = """
            amiq_eth_com_bridge_dma_config_seq dma_config_seq = amiq_eth_com_bridge_dma_config_se\
            q::type_id::create($sformatf("dma_config_seq"), env.config_sqr);  \
            void'(dma_config_seq.randomize() with {max_size_buffer == {{max_size_buffer}}; \
                write_through == {{write_through}}; interface_used == {{interface_used}};});  \
            dma_config_seq.start(env.config_sqr)
            """;
};
```

Figure 4. Fully defined action

We can see that the arguments defined are typical fields that can be found in a UVM sequence, on a higher abstraction layer, that would not dictate the full behavior of the stimuli but would steer it in a certain direction. The variables inside an action do not have to contain the entire randomization space of the SV/UVM sequence, just the necessary variables for creating the entire scenario collection defined. A larger collection of fields can be present in the UVM/SV test-bench and can benefit from randomization there. All fields that would normally be randomized on the top layer of the UVM hierarchy should be present in the PSS model. The variables present in the body of an action define the full control we have over the functionality of the stimuli characterized by said action.

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 wide 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.

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 PSS a powerful mechanic is the inference. This mechanic allows us to set dependencies between sequences by using "state" variables instantiated in the component and "inputs" and "outputs" of the actions. In Figure 4 the state of the DMA configuration is defined as an "output" of the "dma_config" action. Traversing this action in a PSS solution would imply that the state of the DMA is set to true, which is done by the constraint in the action.

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

```
component amiq_eth_com_bridge_config_comp {
    //Configuration for the DMA block ;
    //Necessary for any kind of traffic
    action dma_config {
        ...
    };

    //Configuration for the MAC wrapper block ;
    //Necessary for sending/receiving data on the external interface;
    //Requires a working DMA for RX traffic and a working Control block and DMA for TX traffic
    action mac_config {
        ...
    };

    //Configuration for the control block ; Necessary for TX traffic;
    //Requires a working MAC for TX traffic and a working DMA for memory reads
    action control_config {
        ...
    };

    //Configuration for the filtering block ; Necessary for filtering illegal RX traffic;
    //Requires a working MAC for RX traffic and a working DMA block for memory writes
    action filter_config {
        ...
    };
};
```

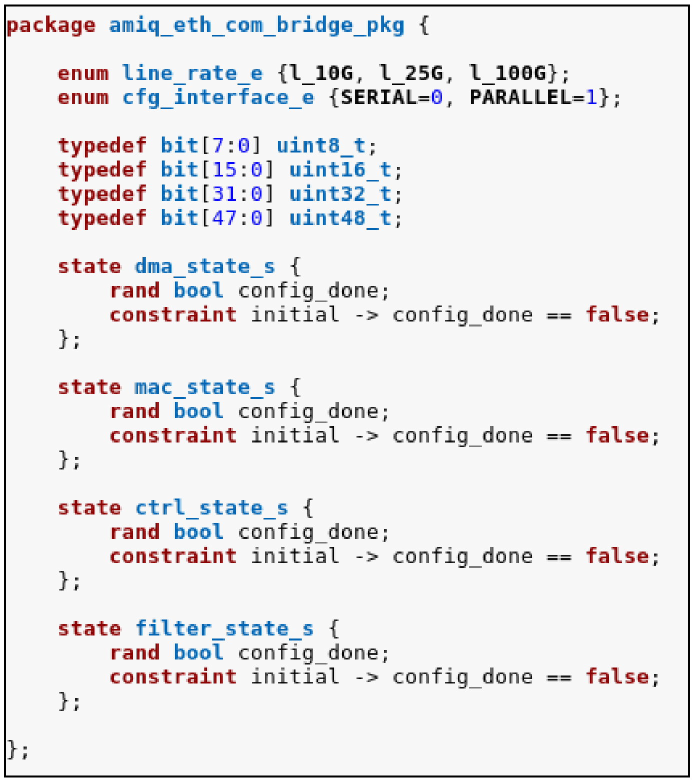
Figure 5. Container with a collection of actions

To encapsulate the entire behavior of the system in response to different configurations being done, the state of all the different configurations should be kept in the top level of the PSS model. For this to be possible, the state variables used as "outputs" of the configuration actions need be defined beforehand. To accomplish this, we define a package that keeps all the type definitions, state variables and resources of the model.

The contents of each state will be composed of the basic variables which constructs the given state as well as constraints for the initial values of these variables. The definition of the state variables for the above component can be seen in Figure 6.

In the case of our system, each state will have a flag which dictates if a certain configuration action has taken place or not. The default value tells us that after reset, none of the configurations took place, so the initial constraints of the flags is "false".

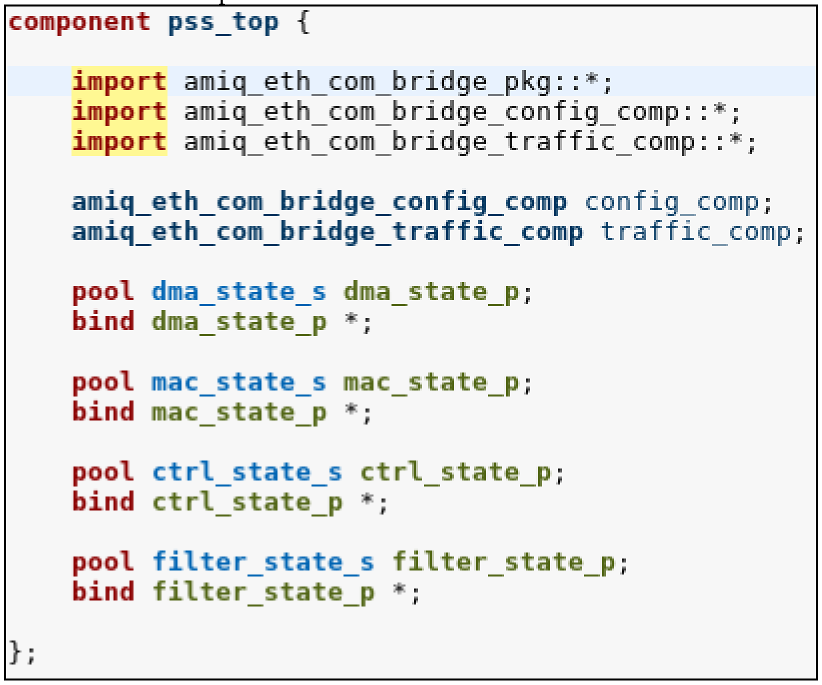
To facilitate this mechanic of action dependencies, inputs and outputs should be set for the consumers and providers of certain states. This is done in a higher layer component which instantiates all the other components and creates the "pools" of states and resources.



```
package amiq_eth_com_bridge_pkg {

    enum line_rate_e {l_10G, l_25G, l_100G};
    enum cfg_interface_e {SERIAL=0, PARALLEL=1};

    typedef bit[7:0] uint8_t;
    typedef bit[15:0] uint16_t;
    typedef bit[31:0] uint32_t;
    typedef bit[47:0] uint48_t;

    state dma_state_s {
        rand bool config_done;
        constraint initial -> config_done == false;
    };

    state mac_state_s {
        rand bool config_done;
        constraint initial -> config_done == false;
    };

    state ctrl_state_s {
        rand bool config_done;
        constraint initial -> config_done == false;
    };

    state filter_state_s {
        rand bool config_done;
        constraint initial -> config_done == false;
    };

};
```

Figure 6. Container component with action sequencing

In our model, the functionality of the TB is encapsulated in two different components, one which dictates what configuration can be applied to the DUT and one that dictates what traffic can be driven. As mentioned above, the pools of the states are defined in the “pss_top” component where all the packages and types are imported together with the instantiations of the available components.



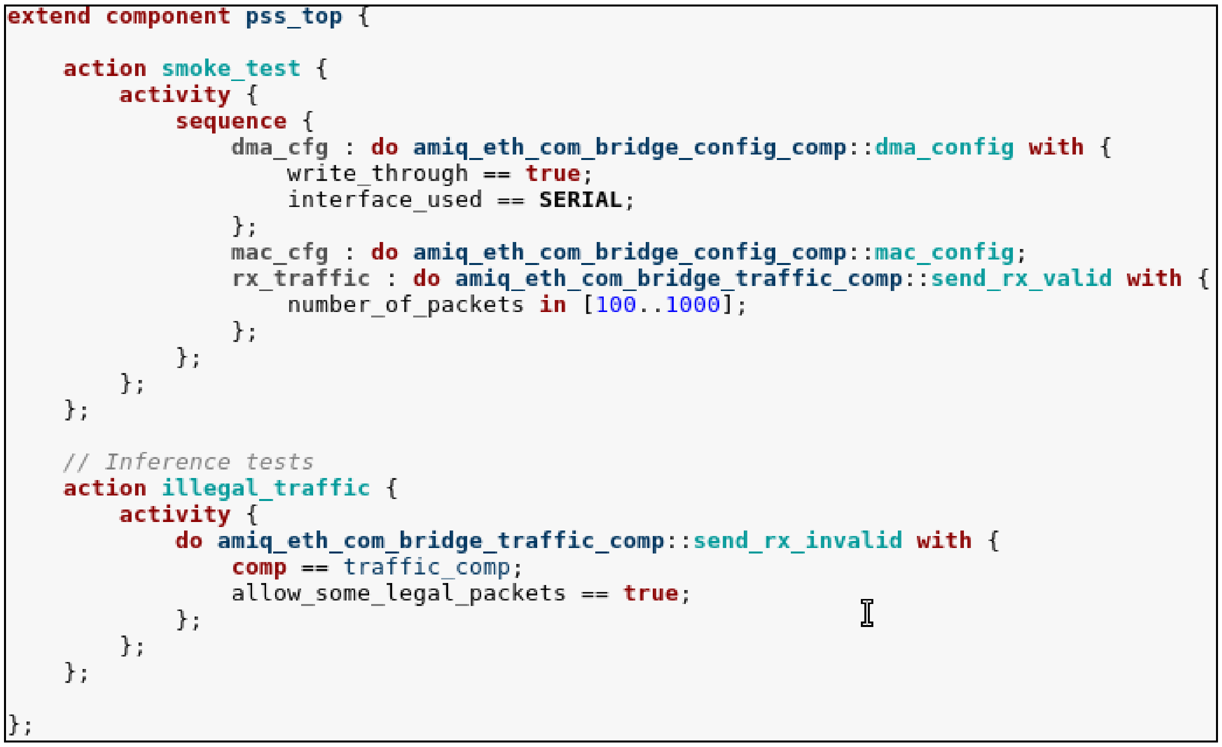
```
component pss_top {

    import amiq_eth_com_bridge_pkg::*;
    import amiq_eth_com_bridge_config_comp::*;
    import amiq_eth_com_bridge_traffic_comp::*;

    amiq_eth_com_bridge_config_comp config_comp;
    amiq_eth_com_bridge_traffic_comp traffic_comp;

    pool dma_state_s dma_state_p;
    bind dma_state_p *;

    pool mac_state_s mac_state_p;
    bind mac_state_p *;

    pool ctrl_state_s ctrl_state_p;
    bind ctrl_state_p *;

    pool filter_state_s filter_state_p;
    bind filter_state_p *;

};
```

Figure 7. Container component with a fully defined knob propagation

By wildcard binding these states, the tools will search in all the components available in the current scope, which is the “pss_top”, and find all the providers and consumers of that particular “state”. The consumers and providers are given by the existence of the “input” and “output” keywords in certain actions.

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 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.



```
extend component pss_top {

    action smoke_test {
        activity {
            sequence {
                dma_cfg : do amiq_eth_com_bridge_config_comp::dma_config with {
                    write_through == true;
                    interface_used == SERIAL;
                };
                mac_cfg : do amiq_eth_com_bridge_config_comp::mac_config;
                rx_traffic : do amiq_eth_com_bridge_traffic_comp::send_rx_valid with {
                    number_of_packets in [100..1000];
                };
            };
        };
    };

    // Inference tests
    action illegal_traffic {
        activity {
            do amiq_eth_com_bridge_traffic_comp::send_rx_invalid with {
                comp == traffic_comp;
                allow_some_legal_packets == true;
            };
        };
    };

};
```

Figure 8. Action that encapsulates a testcase with control knobs and randomization

The last and highest layer of the PSS model is the action which defines the testcase. The PSS model allows for basic extension of existing components which translates into the code defined in the extension to be appended to the initial declaration of said component. In Figure 8, we extend the “pss_top” component and define a couple of scenarios. One of the requirements we have on the model is to be able to define directed scenarios. One such example is the “smoke_test” action.

In the directed testcase, we aim to first configure the DMA, followed by the MAC configuration, and then start driving valid RX data. In this case, we want to see the configurations happen serially and subsequently, the traffic, which requires both a DMA and a MAC to be configured, must be chained serially as well.

The result of running this action with a PSS engine can be seen in Figure 9. Even though the testcase is directed, a collection of randomizable variables which can have an impact on the system functionality is defined in both the configuration, as well as the traffic actions. Multiple generation of the scenario will yield different results on how the variables are generated.

Because our goal is to generate a finite number of scenarios, constraints must be applied on all the variables that are fixed and the “fill” mechanic of the PSS tools can be used to generate all of the scenarios with valid combinations of the variables chosen to be “filled”.

To assess the second defined scenario, done with the inference method, we need to look at how the “send_rx_invalid” action looks like. The short definition of the action can be seen in Figure 10.

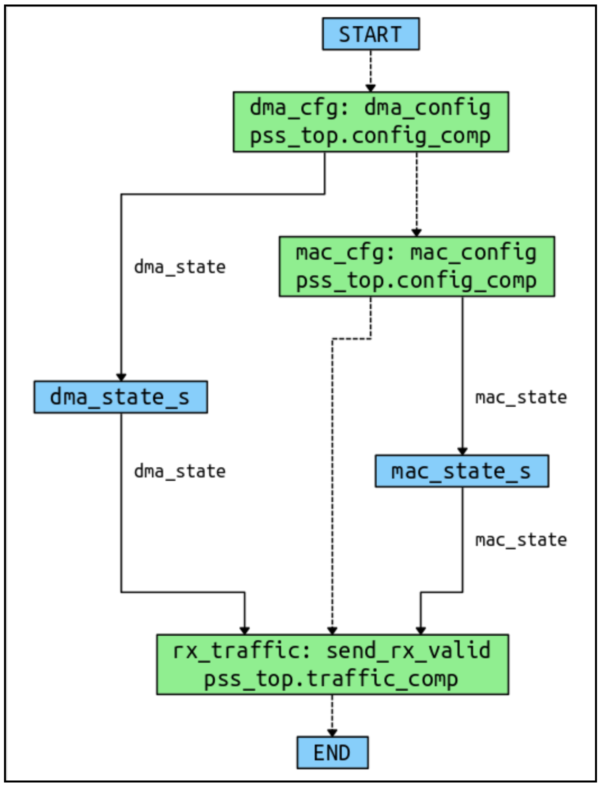
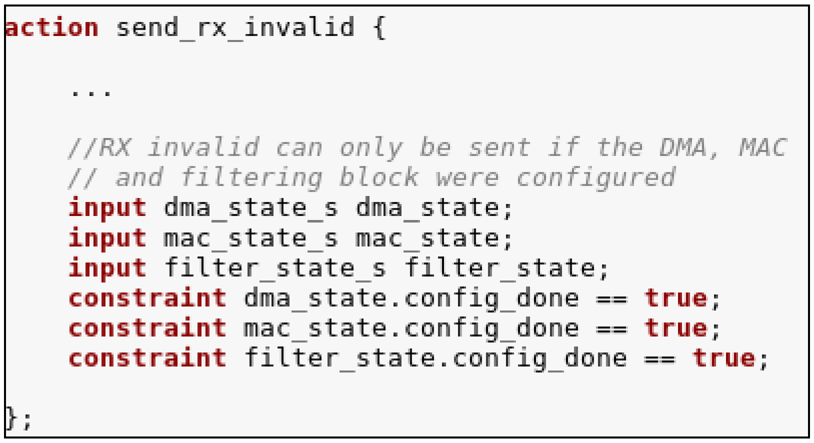


Figure 9. Directed testcase generation

As stated earlier in the configuration component, three different blocks are necessary to be configured before the action can be used in a correct fashion. By telling the tool, that we want this action to be present in the scenario, it will automatically find a way of making this happen.

Furthermore, by generating testcases based on that scenario multiple times, the same action will result in different activity diagrams exploring diverse ways through which the objective of correctly driving illegal traffic can be achieved. Once more, all the fixed variables, which does not affect the testcase should be constraint and all the relevant variables should be "filled" to achieve the test plan goals.



```
action send_rx_invalid {

    ...

    //RX invalid can only be sent if the DMA, MAC
    // and filtering block were configured
    input dma_state_s dma_state;
    input mac_state_s mac_state;
    input filter_state_s filter_state;
    constraint dma_state.config_done == true;
    constraint mac_state.config_done == true;
    constraint filter_state.config_done == true;

};
```

Figure 10. Action with a collection of dependencies on different DUT states

In Figure 11, the outcome of generating scenarios from the "illegal_traffic" action is visible. The sequencing of the configuration actions, as well as some their internal variables, are not set to any values. This allows the tool to explore different valid combinations, given by the constraints set in the action, as well as the dependencies created between the lower layer actions.

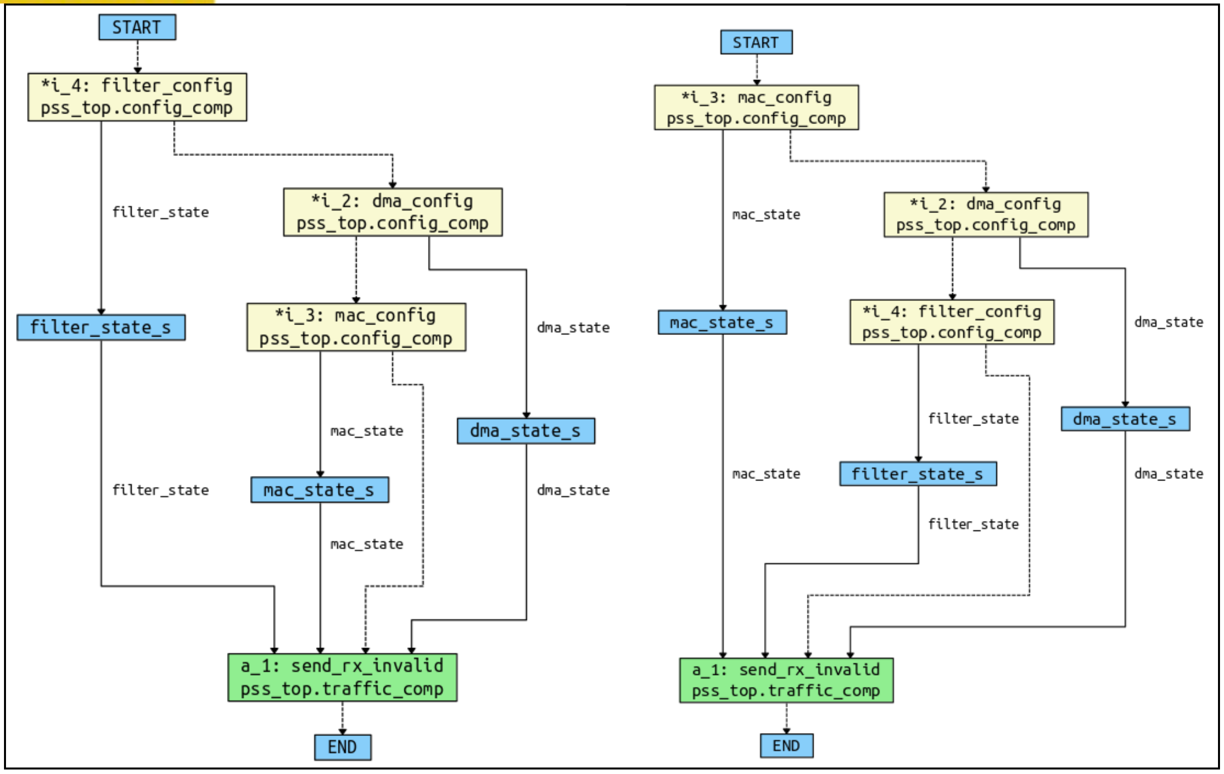


Figure 11. Different activity diagrams obtained from the same action via inference

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 we 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.

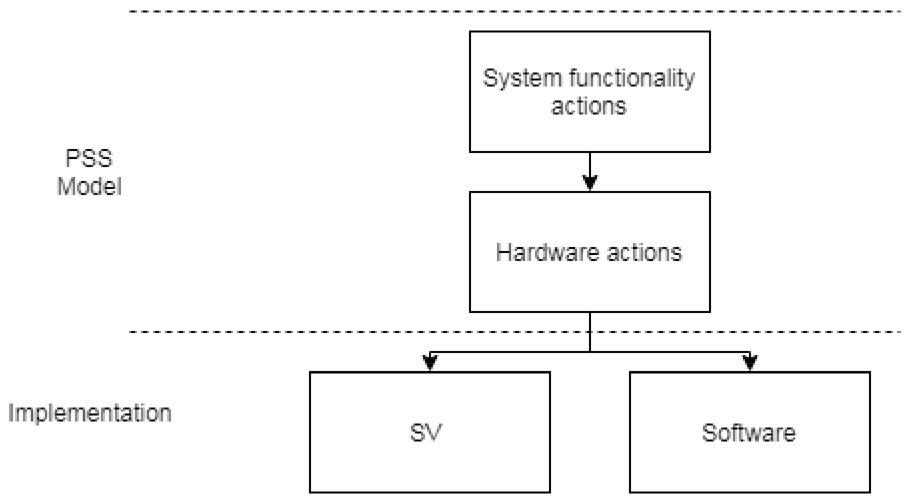


Figure 12. PSS Model layering for multiple platforms

On the first tries we did after implementing the PSS model, some of the testcases generated by the tool had incoherencies. Because we had diagrams provided for all the generated testcases, we were able to spot and correct the problems before even running them in the SV environment.

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.

In Figure 12 the concept of action layering is displayed. As sequence layering is used in UVM/SV to build reusable logic by propagating constraints with distinct levels of abstraction, PSS actions can be structured the same way. In our case, high level network scenarios can be defined as the backbone of our DUT which can be broken down in hardware stimuli which subsequently will translate to different stimuli depending on the platform that the testcase is going to run.

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