Scalable Functional Verification using PSS

Santosh Kumar, Yogish Kumar Raja, Geetika Agrawal,
Karthikeyan S, Arjun Ashok V, Tommy Brunansky
Qualcomm Technologies Inc.

2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES
SAN JOSE, CA, USA
MARCH 4-7, 2024

Problem Statement

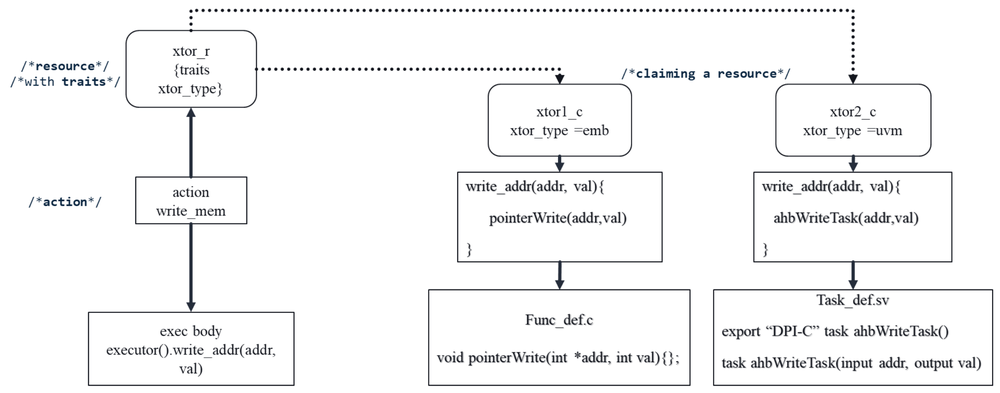
- Complex designs and diverse chip architectures require multiple functional verification strategies to meet fast time to market as well as high design quality.
- This would often involve verification in multiple modes and multiple design integration levels involving processor based stimulus or BFMs driving them or a combination of both.
- Multiple platforms namely simulation, emulation and prototyping boards are used based on speed, controllability, realism and practicality.
- Each such verification environment requires stimulus to be written in a compatible language and methodology
- Some platforms traditionally do not support ability to use constraints and randomization.
- This introduces barriers to adopting a new environment or platform and also significantly increases effort in re coding the stimulus to specific requirements.

Solution and Focus of this paper

- Portable Stimulus and Test Standard (PSS) is a standard language from Accellera that specifically tries to address the above problem
- It provides the constructs to achieve attribute as well as scenario and scheduling randomization , action inferencing and resource and memory management.
- It allows an abstraction layer to help specify stimulus independent of platform and then generate code targeting specific platform
- PSS along with the methodology we have implemented in Qualcomm allows us to build a scalable verification environment.
- The Paper however focusses on a specific part of this overall solution.
  - Use of PSS on top of UVM agents and BFMs
  - Mixed mode use controlling of embedded processor stimulus and interface driven stimulus
  - Constructs PSS provides and techniques we have employed to address core to top reusability

Key PSS constructs and Executor mapping

- Key constructs relevant are components, actions, resource objects, executors and traits
- Actions serve to decompose scenarios into elements whose definitions can be reused in many different contexts
- Executors in PSS are used to represent embedded cores or UVM agents that actions can be scheduled on and traits used to differentiate executor instances
- Using a DPI-C layer serves 2 purposes
  - Action with a single exec body using native execs can be mapped to different implementations based on executor type
  - Generated tests do not require re-elaboration of the TB which would be the case if generating native SV code



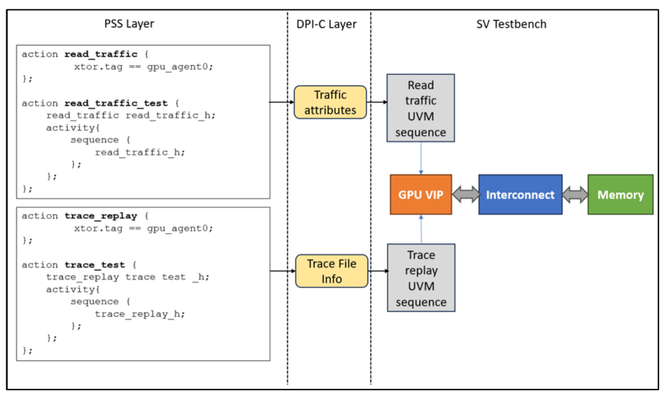
Use Cases Tested

```
function void config_function(init_s s, xtor_trait_s xtor_trait);
import target C function config_function;

extend action config_IP0 {
    exec body {
        config_function(s,xtor.trait);
    };
};
```

```
//Extern functions are SV DPI-C exports
extern void sv_config_function(init_s s, xtor_trait_s xtor_trait);
extern void sv_set_agent_active(xtor_trait_s trait);
extern void sv_set_agent_inactive(xtor_trait_s trait);

void config_function(init_s s, xtor_trait_s xtor_trait){
    sv_set_agent_active(xtor_trait);
    sv_config_function(s,xtor_trait);
    sv_set_agent_inactive(xtor_trait);
};
```

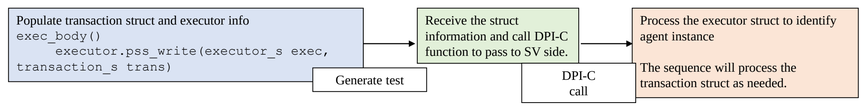
Case1: To speed up bringing up resources on a SoC, multiple SV agents can run initialization sequences faster and parallelly. An AHB agent takes control of the interface dynamically to perform clock and chip resources bring up and relinquish once done

```
assign (supply1,supply0)  `NOC_HIER.noc_hwdata
= agent_active ? agent_if.hwdata : 'bz;
```



Case 2: For Performance verification of interconnects, we run workloads on a graphics processor. But this does not require the actual graphics processor RTL. By using a trace replay sequence scheduled on GPU SV agent (this is a BFM hijacking the output interface of the processor) as an executor we have a lightweight method of mimicking the workload on the interconnect.

Use Cases Tested



Traffic fields in the transaction_s struct to control on the interface

```
struct transaction_s {
     rand bit[31:0] addr,
     rand bit[255:0] data,
     rand bit[5:0]opcode,
     rand bit[2:0] length,
     rand bit[2:0] cacheability ,
     rand bit write_allocate_policy
};

//definition for <PROT> executor type
<prot>_pss_write(executor_s exec,
transaction_s trans)
```

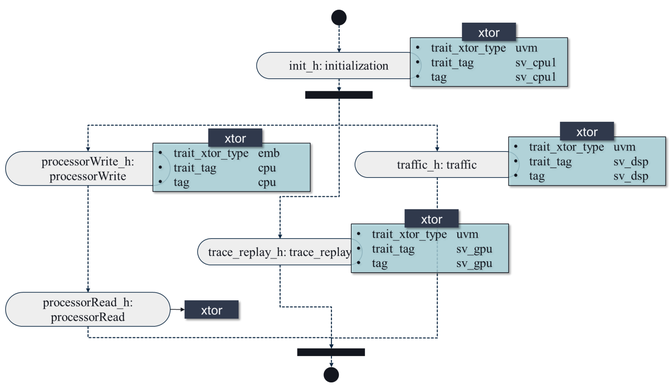
```
task sv_<prot>_write_task(executor_s exec,transaction_s
traffic);

     //Step 1: retrieve <prot> sequencer handle
    my_sequencer_h = get<prot>SeqHandle(exec.inst);
     //Step 2. Insantiate sequence with constraints

    <prot>_write_sequence my_seq_h;
    if(!my_seq_h.randomize() with {
    //populate with traffic struct
            my_seq_h.addr == traffic.addr;
        my_seq_h.data == traffic.data;
        my_seq_h.opcode == traffic.opcode;
    }
     //Step 3: start sequence on sequencer
    my_seq_h.start(my_sequencer_h);

endtask
```

Case 3: I/O Coherency verification requires both the realism of an actual CPUs with caches that need to be snooped and a I/O coherent master like a DSP pumping in coherent traffic. The DSP can be replaced with an AXI/Custom Bus agent to have better control on the low level attributes driven on the bus while performing shared data operation with the CPU.
Here Each protocol agent type is represented as an executor type allowing mapping of exec body to respective DPI-C function and corresponding SV task.

Complete SoC level Scenario example

```
action scenario {

  initialization init_h;
  write_read_seq wr_rd_seq_h;
  trace_replay trace_replay_h;
  traffic traffic_h;
  constraint wr_rd_seq_h.xtor.tag == sv_gpu;
  constraint trace_replay_h.xtor.tag == sv_gpu;
  constraint traffic_h.xtor.tag == sv_dsp;

  activity{
    sequence{
      init_h;
      parallel{
          wr_rd_seq_h;
          trace_replay_h;
          traffic_h;
          };
      };
    };
  };
```

- init_h : An action handle that signifies initialization of the system before the scenario can be run
- Wr_rd_seq_h: An action handle that represents an embedded processor writing and reading form memory
- Trace_replay_h: an action that helps drive a random or a pre generated traffic pattern representing a GPU workload
- Traffic_h: Action that helps drive coherent transactions either from an embedded DSP core or a BFM replacing the processor.



Conclusion

With these approach for verification, we can scale a given PSS scenario specification across different functional verification strategies - "**One Stimulus to rule them All**". Following are some of the key benefits and results

(i) Quick and easy scenario creation
  a. Enabling easy plug-play playground without worrying about synchronization or data flow between processors, between agents or between processors and agents as well.

(ii) Gen-time Functional coverage metrics
  a. Scenario is solved and graph generated prior to simulation thus enabling coverage to evaluate scenarios even before running them.

(iii) Fast initialization
  a. We observed improved runtimes (~30% on average) improving the reducing the init-to-test ratio.

(iv) Improved verification Coverage
  a. With a mix of controllability using agents and realism of actual processor RTL corner case bug hunting is easier

References

1. PSS2.0 LRM :https://accellera.org/images/downloads/standards/Portable_Test_Stimulus_Standard_v20.pdf
2. ANSI/IEEE 1800-2012 – IEEE Standard for SystemVerilog—Unified HW Design Specification and Verification Language
3. IEEE 1800.2-2017 – IEEE Standard for Universal Verification Methodology Language Reference Manual
4. Chris Spear , Greg Tumbush, SystemVerilog for Verification, 3rd ed., Springer, 2012.

Contact :
Santosh A Kumar Email: kumarsan@qti.qualcomm.com
Yogish Kumar Raja Email: yraja@qti.qualcomm.com

Acknowledgements:
Thanks to support from our PSS tool partners who have helped with different aspects of our work
Cadence Perspec System Verifier
Synopsys VC Portable Stimulus