Scalable Functional Verification using Portable Stimulus Standard

Santosh A Kumar
Qualcomm Technologies Inc.
kumarsan@qti.qualcomm.com

Geetika Agrawal
Qualcomm Technologies Inc.
geetagra@qti.qualcomm.com

Arjun Ashok Vazhayil
Qualcomm Technologies Inc.
aashokv@qti.qualcomm.com

Yogish Kumar Raja
Qualcomm Technologies Inc.
yraja@qti.qualcomm.com

Karthikeyan Sugumaran
Qualcomm Technologies Inc.
kartsugu@qti.qualcomm.com

Tommy Brunansky
Qualcomm Technologies Inc.
tbrunans@qti.qualcomm.com

Ever increasing design complexity across different market segments (Auto, Mobile, Servers, Compute etc.) and different architecture types (single die vs chiplets) has put verification effort and strategies used across IP, Subsystem and SOC under the spotlight. With challenging TTM (time to market) for products, it is imperative to have a scalable verification approach that allows single constrained random stimulus specification to be reused across different verification environments and strategies.

In this paper, we will present our experience of using Portable Test and Stimulus Standard (PSS) language to enable seamless reusability of constraint random scenarios across platforms, design integration levels and verification environments.

I. PROBLEM STATEMENT

A complex design undergoes layered verification from Block/IP level to a bigger Sub-System Level and eventually to an SoC. Further complexities get introduced at the SoC level based on whether it is a monolithic SoC vs multi-die/chiplet based design. Based on the design features, scenario requirement and complexity, different design integration level and verification platform strategies are applied to functionally verify the design. Each environment and level of design integration offers its strengths in terms of speed, controllability, realism, and practical coverage scope. Currently stimulus needs be coded based on the above choices. The language used to formally describe the scenario also varies from one environment to the other. IP/Block level verification often use System Verilog language and UVM for constrained random verification. Sub systems and SoCs use a mixture of environments and stimulus specifications especially when processor IPs are involved. Fast platforms/prototyping platforms mostly rely on C based stimulus.This introduces barriers for core to top reuse, adds manual effort needed to tailor the same scenario to multiple platforms and environments and reduces the number of functional verification cycles that can be spent on bug hunting.

II. PROPOSED SOLUTION

PSS provides the means to model constrained random scenario sequences and use them as an abstraction layer between scenario specification and scenario implementation. The PSS model and tool manages attribute level constraints, resource and memory management, data flow attributes that can be randomized based on constraints, and executors that can be assigned to carry out actions. The solved scenarios are then mapped to executables that are run on either real processors or BFMs (UVM agent). The remainder of the paper discusses some of the building blocks using PSS to help achieve this goal and a few case studies using this approach.

III. INTRO TO PSS CONSTRUCTS

Components: Components serve as a mechanism to encapsulate and reuse elements of functionality in a portable stimulus model. Typically, a model is broken down into parts that correspond to roles played by different actors during test execution. Components often align with certain structural elements of the system and execution environment, such as hardware engines, software packages, or testbench agents.

Actions: Actions are a key abstraction unit in PSS. Actions serve to decompose scenarios into elements whose definitions can be reused in many different contexts. Along with their intrinsic properties, actions also encapsulate the rules for their interaction with other actions and the ways to combine them in legal scenarios. Atomic actions may be composed into higher-level actions, and, ultimately, to top-level test actions, using activities.

Resource objects: Resource objects represent computational resources available in the execution environment that may be assigned to actions for the duration of their execution. Resource objects may be locked or shared by actions.

Executors: A PSS generated test calls foreign functions available in the target environment, executes target-language code blocks, and performs target operations provided in the core-library. It does so in accordance with the user-defined realization of actions and of flow/resource objects specified in the form of target exec blocks— body, run_start, and run_end—and functions called from them. Foreign function calls, target-language code blocks, and built-in target operations, all need to be performed under a certain agent of execution available to the test in the runtime environment, or in short, an executor.

Traits: The PSS core library provides means to represent executors in the PSS description and to assign scenario entities to them. Executors are characterized by user-defined properties called traits, which serve to control the assignment of actions/objects to them. For example, the cluster of a CPU core could be represented as a trait attribute. Related executors are grouped together so that scenario entities can be assigned to a random instance out of a group. The selection of executors satisfies constraints on their trait attributes if any are specified.

IV. PSS LAYERING OVER UVM OR VANILLA SYSTEM VERILOG

Below is a technique we have employed using *"executors"* in PSS to differentiate, assign and customize functional code based on the appropriate execution entity in the DUT/TB.

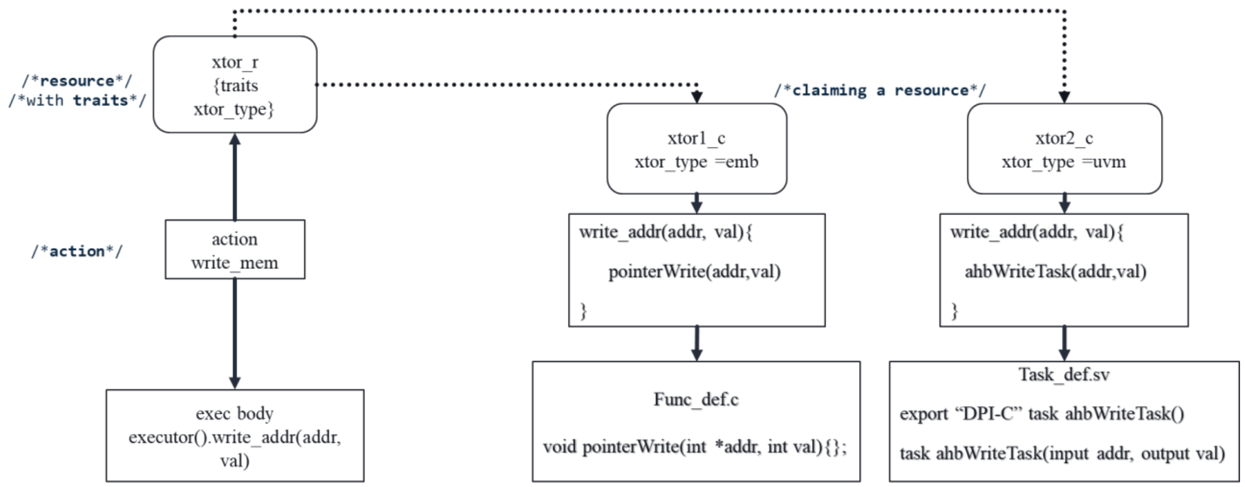


Figure 1: PSS Executor mapping example

In the above example we show one of the abstraction techniques that PSS provides and the one we predominantly use in this paper. The executor component in PSS represents either an embedded physical core in the design or it could also be a TB component such as a UVM agent that is capable of driving transactions to the DUT or any interface on the DUT independently. Different embedded core types or UVM agent types are represented as different executor types in our PSS model. Multiple cores or agents of the same type are represented as different instances. All the executor component types are however templatized using the same trait struct. This allows all the executor component instances irrespective of the type to be added to a single executor_group. The action consumes an executor resource using a lock. The lock ensures that only one action can consume a particular resource instance from the pool at a given point in the execution. The number of resources in the pool of executor resources are the same as the number of executor component instances in the executor_group. The executor resource xtor_r is templatized using the same trait struct that was used for the executor component types. Based on the constraints on the trait attributes on the executor

resource a corresponding executor component instance is allocated. We have multiple traits in trait struct and a combination of 3 traits in our system can be used to uniquely allocate a particular executor instance if needed.
Each executor type can implement the same functions differently. In the example in Figure 1 write_addr() is a function that is implemented differently for each executor type. The action always calls write_addr() with the solved values of address and data but based on the executor the corresponding implementation is used. We use the PSS native execs for our exec body, and this allows us to use the same imported function call but mapping to different implementations based on the executor chosen.

```
struct xtor_trait_s : executor_trait_s {}

resource xtor_r : executor_claim_s<xtor_trait_s> {};

component xtor_c : executor_c<xtor_trait_s>{};
component emb_xtor_c : xtor_c{};
component uvm_xtor_c : xtor_c {};
component cpu_core_c : emb_xtor_c {};
component cpu_chi_agent_c : uvm_xtor_c {};

component pss_top {
        cpu_core_c cpu0;
        cpu_chi_agent_c chi0;

        exec init_down {
                xtor_group.add_executor(cpu0);
                xtor_group.add_executor(chi0);
};
};
```

Figure 2: Code snippet for executor inheritance and registration

V. FAST INITIALIZATION USING PSS LAYER

Scenarios might require agents to temporarily take control of the interface and drive transactions and then return control back to the RTL. This might be needed to create very synthetic corner case scenarios or even run initialization sequences of IPs using agents that are attached closer to the block than using LD/ST instructions of a processor. In this paper we are trying to speed up initialization time by using light weight UVM agents to front door initialize the system in a large design. This could be achieved backdoor as well if the initialization flow allows it. This concept by itself is not new at all but following are some of the challenges that we have been able to resolve using a PSS layer far more seamlessly than traditional methods.

1. All initializations cannot be done at time 0 or at the same fixed time in the simulation.
2. Some of the initializations have sequencing dependencies.
3. Initialization parameters and sequences often depend on the randomized scenario to be run.

By creating a scenario where the initialization is also represented by PSS actions, we were able to offload the execution to the corresponding uvm agents. Using scheduling constraints available in PSS we could specify which initialization actions could run in “sequence” and which ones could run in “parallel”. We could also select the agents that would be running these transactions based on the “xtor” assigned to the action. In a simulation environment or a fast platform that supports such agents this could be a uvm_agent while in platforms where such an agent is NOT available one could just change the xtor to pick an executor of embedded core type.
To use an SV Agent to temporarily take control of the interface that it is driving we must introduce tristate buffers in the TB in order shift control between agent and the CPU path. Control of the tri-state buffers to pick the right initiator is handled by the PSS scenario executors and scheduling constraints. In the example below when an action is assigned an executor that maps to an AHB agent, its corresponding task sets *agent_active* to 1 at the start of the task and sets it back to 0 before return. Here the AHB agent used is a standard off the shelf UVM VIP.

```
enum clk_speed_e {TURBO,NORMAL,LOW};
extend component pss_top{
   action base_action { lock xtor_r xtor;};
   struct init_s {rand clk_speed_e clk_speed;};

   action config_IP0 : base_action {rand init_s s;};
   action config_IP1 : base_action {rand init_s s;};
   action config_IP2 : base_action {rand init_s s;};

   action initialization {
       rand init_s top_config;
       config_IP0 config_IP0_h;
       config_IP1 config_IP1_h;
       config_IP2 config_IP2_h;

       constraint config_IP0_h.xtor.tag == agent0;
       constraint config_IP1_h.xtor.tag == agent1;
       constraint config_IP_h.xtor.tag == agent2;
       constraint forall (s : init_s){
          s.clk_speed == top_config.clk_speed;
       };

       activity{
           sequence{
               config_IP1_h;
               parallel{
                   config_IP2_h;
                   config_IP0_h;
               };
           };
       };
   };
};
```

```
enum mode_e {SLOW, FAST};
action my_scenario {
    rand mode_e mode;
};

action test1 {
    initialization initialization_h;
    my_scenario my_scenario_h;

    constraint my_scenario_h.mode ==
FAST ->
initialization_h.top_config.clk_speed
==TURBO;

    activity{
        sequence {
            initialization_h;
            my_scenario_h;
        };
       };
};
```

Figure 2: Code Snippet for initialization sequence and subsequent use in a test along with a scenario.

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

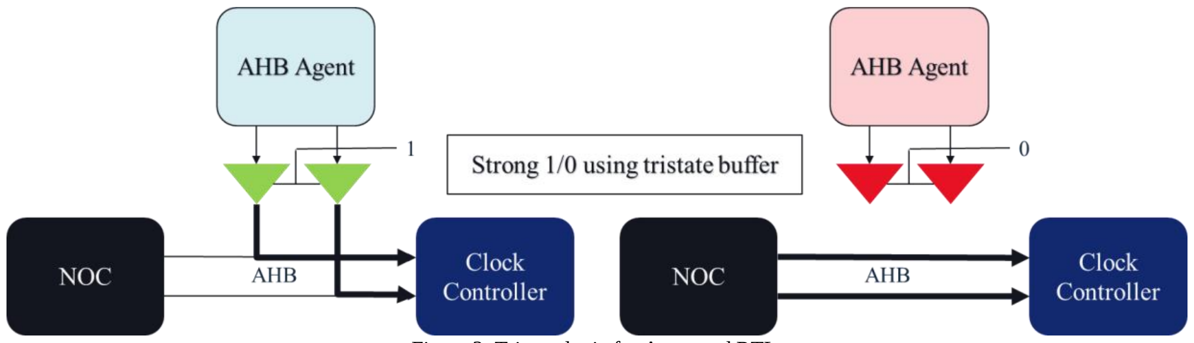


Figure 3: Tristate logic for Agent and RTL.

The timing of driving the agent_active signal is controlled by the sequencing of the actions. This also ensures that the control is taken in a safe point in the simulation. Also, the scheduling constraints can be used to ensure no conflicting action is executed that may affect or be affected by the agent temporarily taking control of the bus. The reason for implementing such a flow is to allow for scenarios where the first-time initialization can be done using an agent while any further dynamic re-initialization during the scenario could be done so by some embedded core reactively.

```
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){
    //saveCurrentScope
    //setNewScope
    sv_set_agent_active(xtor_trait);
    sv_config_function(s,xtor_trait);
    sv_set_agent_inactive(xtor_trait);
    //setOriginalScope
};
```

```
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);
    };
};
```

Figure 4 : Agent control synchronization

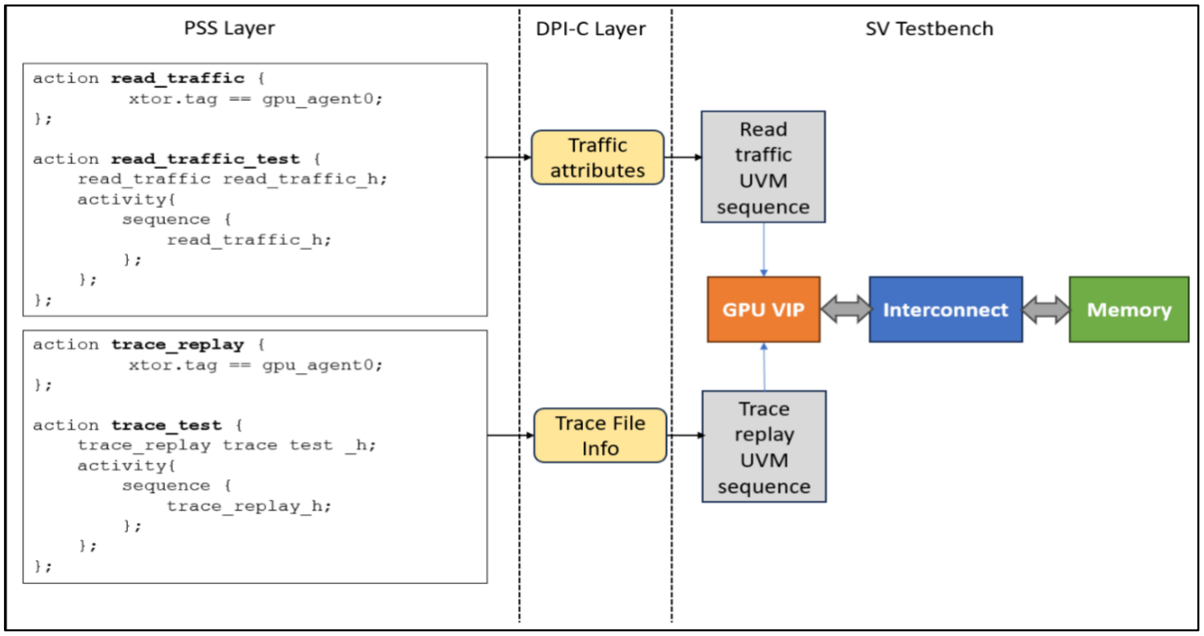
VI. SOC LEVEL SCENARIO CASE STUDIES

Following are some interesting real world case studies performed at SOC Verification using the techniques discussed in the previous section

Case 1: To accelerate bringing up resources on a SoC, multiple SV agents can run initialization sequences faster and parallelly. An AHB agent takes control on the bus using the aforementioned tristate logic to perform clock and chip resources bring up as explained in Section 5.

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.

In such scenarios where interconnects and memories are the primary focus of verification, we remove dependency on the actual RTL of the initiator to be available and configured during simulation. More importantly, this form of stimulus enables full control on the traffic pattern including bus attributes such as address, data length, memory attribute, opcode, priority level, etc., and even sequence timing parameters like transaction delays. PSS allows constraining the above attributes during scenario creation for generating custom synthetic traffic or directly reuse trace files generated by other teams to mimic specific use cases to achieve comprehensive performance coverage of the SoC fabric and memory. Moreover, with the scheduling capabilities of PSS, the ability to create concurrent traffic scenarios utilizing multiple UVM initiators and embedded processors becomes effortless while maintaining controllability and randomness on the stimulus. This hybrid testbench approach for verification is immensely powerful by tailoring scenarios which target specific issues or introduce randomness to aid in bug hunting..



```
action read_traffic {
        xtor.tag == gpu_agent0;
};

action read_traffic_test {
    read_traffic read_traffic_h;
    activity{
        sequence {
            read_traffic_h;
        };
    };
};
```

```
action trace_replay {
        xtor.tag == gpu_agent0;
};

action trace_test {
    trace_replay trace test _h;
    activity{
        sequence {
            trace_replay_h;
        };
    };
};
```

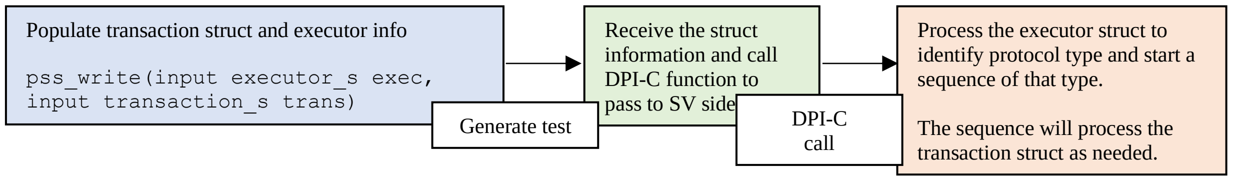
Figure 5: GPU SV agent controlled using different stimulus formats from PSS with the same underlying testbench through a DPI-C layer.

In the above example read_traffic action represents an example where the PSS code controls the transactions on the fine grain level while the trace_replay action helps randomize which pre-generated trace file that contains a transaction queue.

Case 3: I/O Coherency verification requires both the realism of 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.

Assuming a scenario of data sharing between CPU (fully coherent) and an I/O coherent initiator where we replace I/O initiator with an agent. A typical I/O initiator interface would have address fields, data fields, control fields like opcode and memory transaction type (various flavors of cache-ability, write allocate and write update policies). A bug can be present for a very specific combination of these various fields. Working on embedded version of the I/O initiator might require programming internal registers for the initiator to drive a certain memory transaction type. There might be some specific instructions that need to be executed by the initiator in order to generate a given opcode. For example, CHI opcode WriteUniquePtl can be generated when the initiator executes a specific type of store instruction. All these limitations can be overcome if we can have control directly over the interface.

In the snippet shown in Fig 6, the retrieval of the AHB sequencer handle is done also using the executor_s struct that is passed to the task from the PSS code. Each agent instance is represented as a unique executor instance in the model and hence the *exec* struct is populated using call to ***executor()*** function in PSS which returns the executor instance assigned to the current action and hence the corresponding traits of that instance. The trait info that is part of the exec struct helps identify a unique agent instance and hence retrieve the corresponding sequencer handle.


Populate transaction struct and executor info
pss_write(input executor_s exec, input transaction_s trans)
Generate test
Receive the struct information and call DPI-C function to pass to SV side
DPI-C call
Process the executor struct to identify protocol type and start a sequence of that type.
The sequence will process the transaction struct as needed.

```
pss_write(input executor_s exec,
input transaction_s trans)
```

executor_s can have information on the protocol type

Like a sequence_item class in uvm, we can define all the traffic fields in the transaction_s struct that we wish to control on the interface. It can look something like:

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

};
```

The various fields of this struct can be set during test scenario generation.

```
transaction_s traffic_info;
traffic_info.opcode="WriteUniquePtl";
traffic_info.cacheability="Write_Thro
ugh_No_Allocate";
```

This struct would then be passed as it is on the c side.

```
task sv_write_task(executor_s exec,transaction_s
traffic);

      case(exec.protocol)
        AHB: ahb_write(traffic);
        AXI: axi_write(traffic);
        CHI: chi_write(traffic);
      endcase
endtask

//sample function
task ahb_write(transaction_s traffic);

      //Step 1: retrieve ahb sequencer handle
      //Step 2. Insantiate sequence and use
transaction struct to constraint transaction items

        ahb_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
```

Figure 7: PSS to UVM connection layer

Putting it all together is a concurrency scenario shown below at the SoC level where CPU and DSP are running a false data sharing scenario while GPU is pumping in traffic to fabric while running a workload. All of them are targeting a common memory. The same technique can be used for running a similar scenario on the interconnect only verification level where even the CPU is also replaced by an agent. When running on a Fast/Accelerated platform one could set the executors as real processors in place of the agents. In all cases the stimulus specification and scenario structure can remain the same at the PSS level. Each agent or processor is represented in the PSS model as a unique *executor* instance. The PSS model and solver also randomizes the memory buffers to be used in the scenario using address spaces and address claims as described in the PSS core library in the LRM. The AMBA agents listed below are off the shelf agents while the custom ones are used for interfaces that are proprietary.

```
action initialization {
     clk_ctl_c::config_clk config_clk_h;
     constraint config_clk_h.xtor.trait.xtor_type == uvm;
     activity {
        config_clk_h;
        };
     };
action write_read_seq {
     processor_c::processorWrite processorWrite_h;
     processor_c::processorRead processorRead_h;
     rand int loop_count;
     activity {
          sequence {
            replicate (loop_count) {
          processorWrite_h;
          processorRead_h;

         };

        };
     };
  };
```

```
action scenario {
   initialization init_h;
   write_read_seq wr_seq_h;
   trace_replay trace_replay_h;
   traffic traffic_h;
   constraint wr_seq_h.xtor.tag == cpu;
   constraint trace_replay_h.xtor.tag == sv_gpu;
   constraint traffic_h.xtor.tag == sv_dsp;

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

Figure 6: PSS code to describe the above scenario.

The following snippet shows the solution graph of the solved scenario action. As one can see the *clock_config* initialization action is scheduled on the *sv_cpu1* which represents a uvm_agent and in our usecase maps to an AHB VIP. The gpu, dsp and cpu are running concurrent traffic to memory representing a false sharing coherency test. The loop count is set to 2 for illustration purposes but can be set to a much higher count based on the platform.

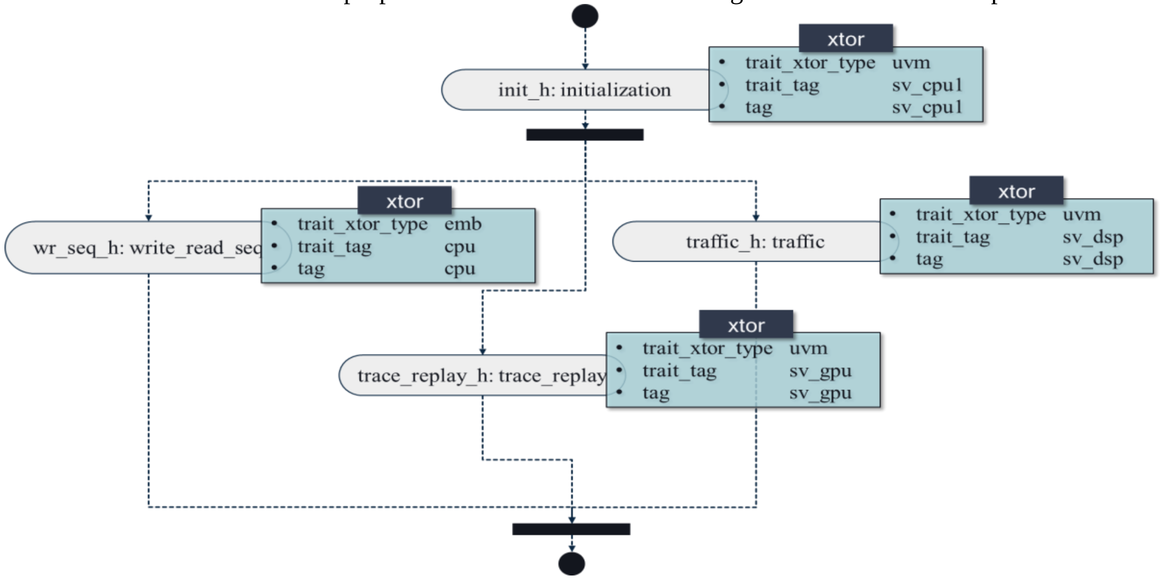


Figure 8: Scenario Solution graph showing concurrent executors.

Note that running the same scenario on a platform that does not support a UVM agent would simply mean changing the *xtor_type* constraint to be *emb*. This would automatically generate code that only uses embedded cores and hence could easily be run on fast platforms that may only support native processor execution.

VII. RESULTS

With these approaches to 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

VIII. 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.