2025
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Expediting Coverage Closure in Digital Verification with the Portable Stimulus Standard (PSS)

Tulio Pereira Bitencourt, Nikolaos Ilioudis, Ahmed Abd-Allah, Daniel Waszczak, Anton Tschank, Tom Fitzpatrick

SIEMENS

accellera
SYSTEMS INITIATIVE

Agenda

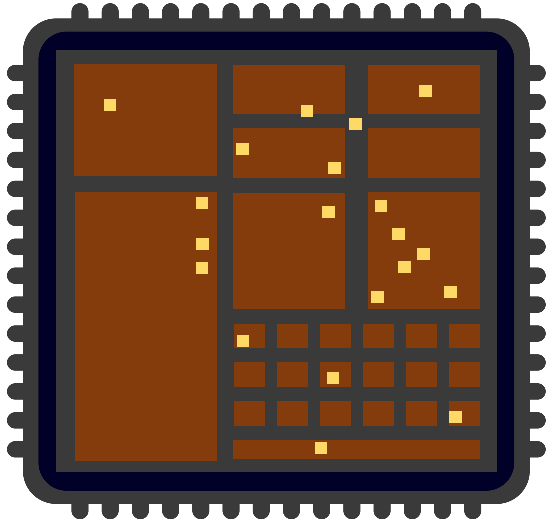
1. Introduction to Portable Stimulus Standard (PSS)
2. Effectively creating a robust PSS model
3. UVM & PSS: a teamwork to expedite digital verification
4. Using PSS from day one
5. Adapting existing UVM testbenches to support PSS
6. Portable Stimulus Standard, emphasizing PORTABLE
7. Real-world metrics on the impact of PSS

Tessent Embedded Analytics functional monitoring

Observing non-intrusively if your SoC behaves as it was meant to

Full visibility into HW/SW interactions in deployed systems enabling optimizations and debugging throughout the entire system lifecycle from lab to deployment

- Real-time debug and trace environment
- Optimize software to achieve better performance and efficiency
- Use historical performance data to inform designs of next generation designs



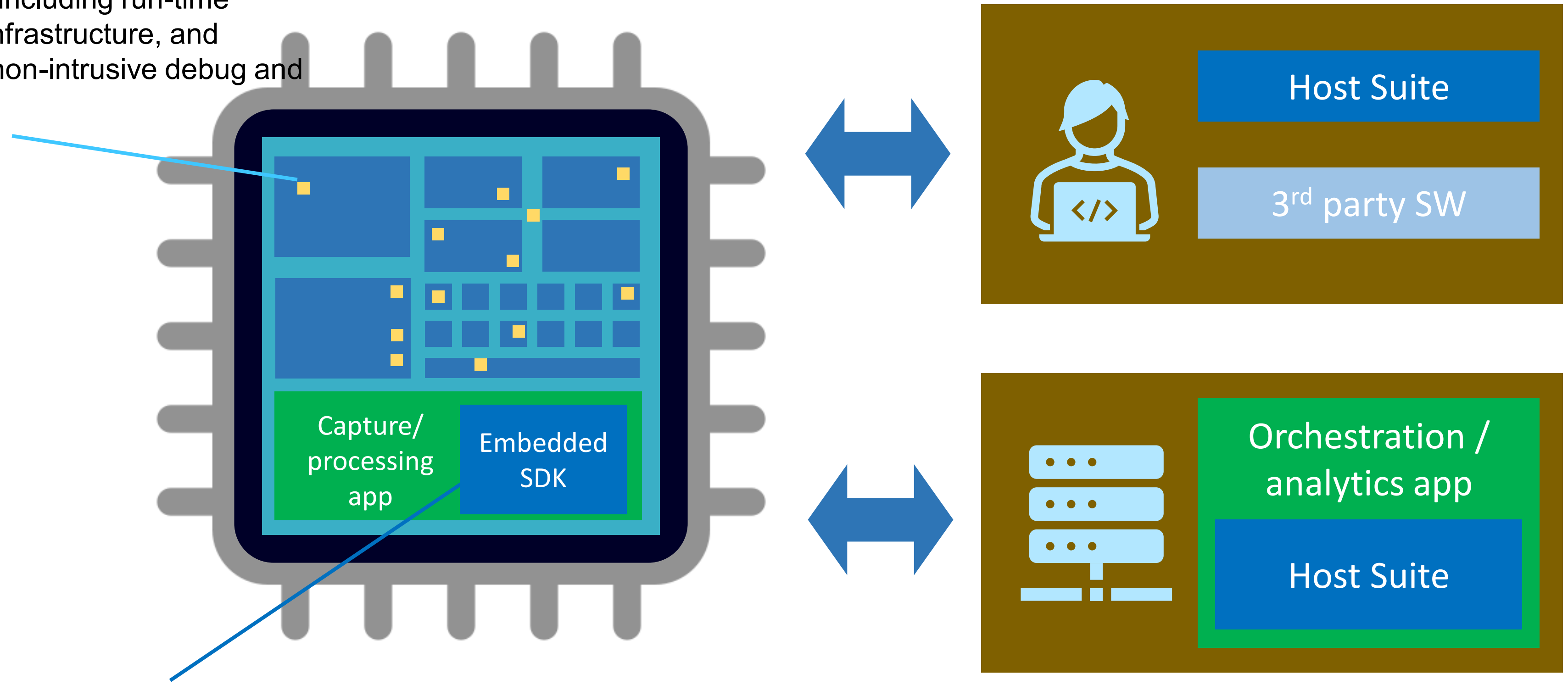
Bus Monitor	Status Monitor	Trace Encoder	Static Instrumen-tation	CPU Debug Module	NoC Monitor	Direct Memory Access	Trace Receiver

AI   Data Centre   Automotive   5G/6G   Storage   Audio

Introduction – Tessent Embedded Analytics

1 **Smart monitors**
Range of ~40 IP blocks including run-time configurable monitors, infrastructure, and interfaces that enables non-intrusive debug and performance monitoring

2 **Software for interactive debug and optimization**
Debug software running on a separate PC is used to interact with the EA smart monitors



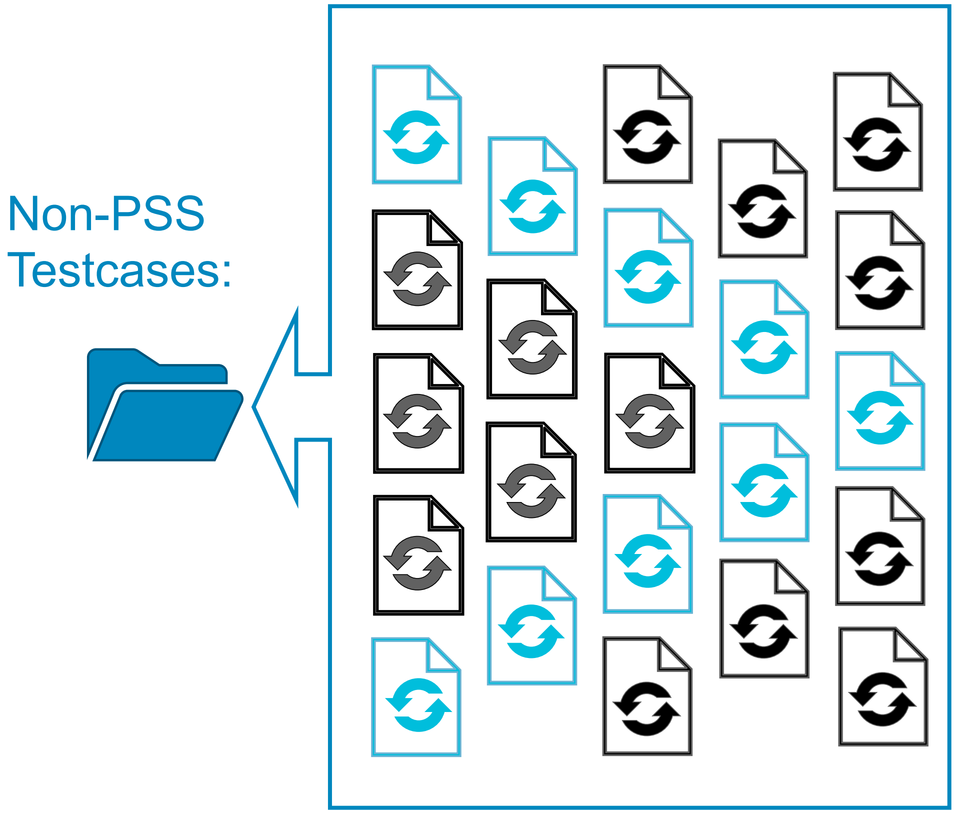
3 **Edge analytics enablement**
Applications developed using the Embedded SDK interact with the monitors, capture, and process results

4 **Fleet monitoring enablement**
Applications developed using Host Suite can automate data orchestration and analytics from one or multiple devices

Introduction to Portable Stimulus Standard (PSS)

Typical UVM Limitations

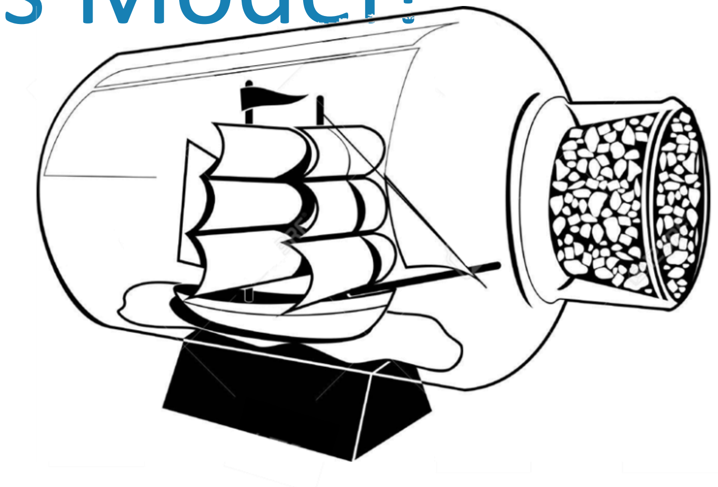
- UVM is a well-stablished technology, but…
  - ➢ Relies on too many files to be managed
  - ➢ Requires the implementation of different testcase scenarios manually – **delaying coverage closure**
  - ➢ Verification intent cannot be ported into non-SystemVerilog solutions



What is a Portable Stimulus Model?

The Abstract Model

- *What* does it do



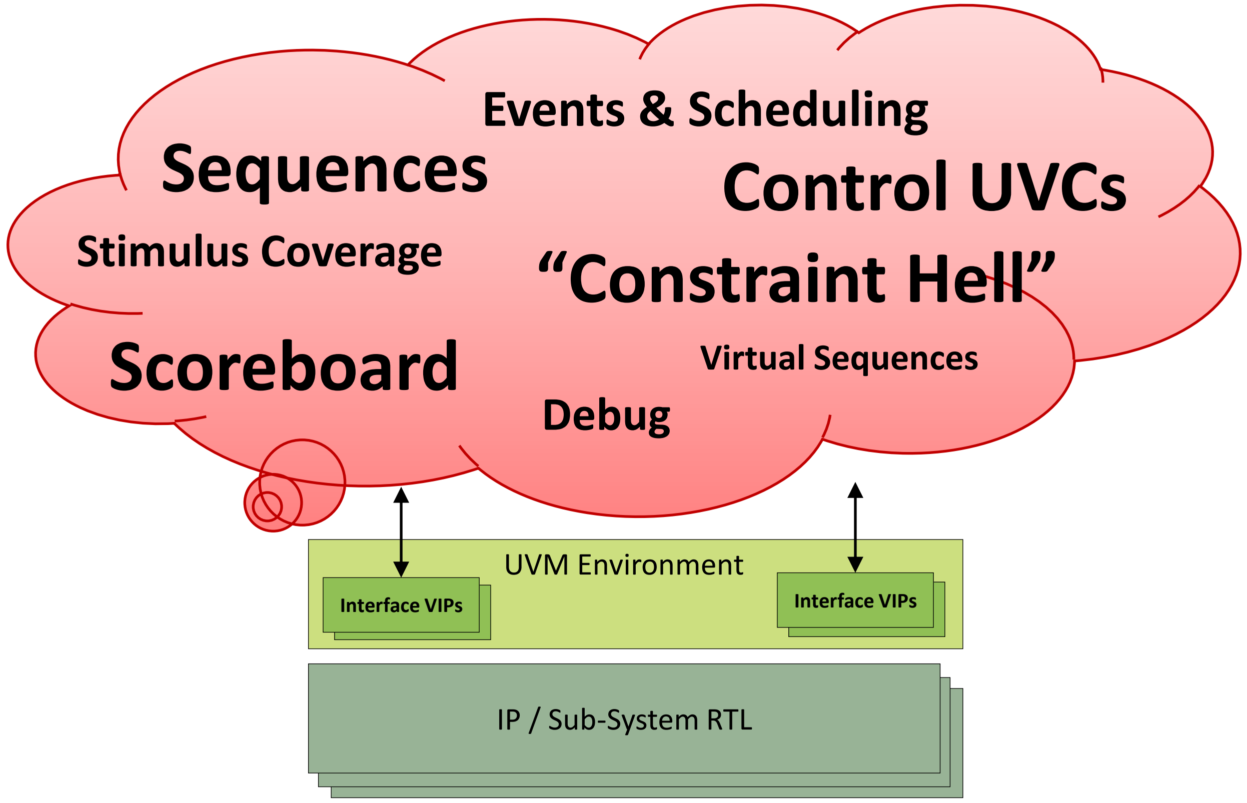
The Realization Layer

- *How* does it do what it does

Concise Language to Specify Verification Intent

A *complement* to UVM, not a replacement

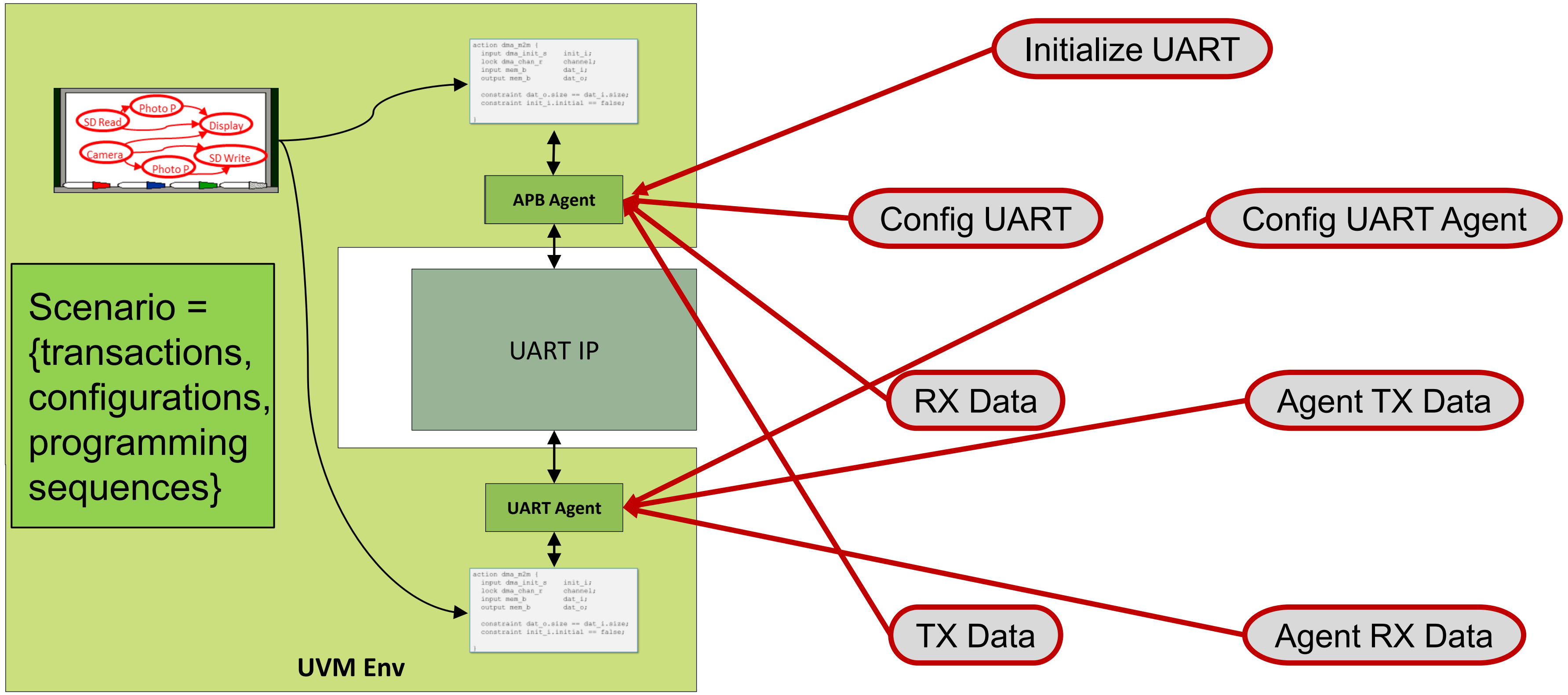
PSS is a stimulus language



Concise Language to Specify Verification Intent

A *complement* to UVM, not a replacement

- Behavior = *Action*



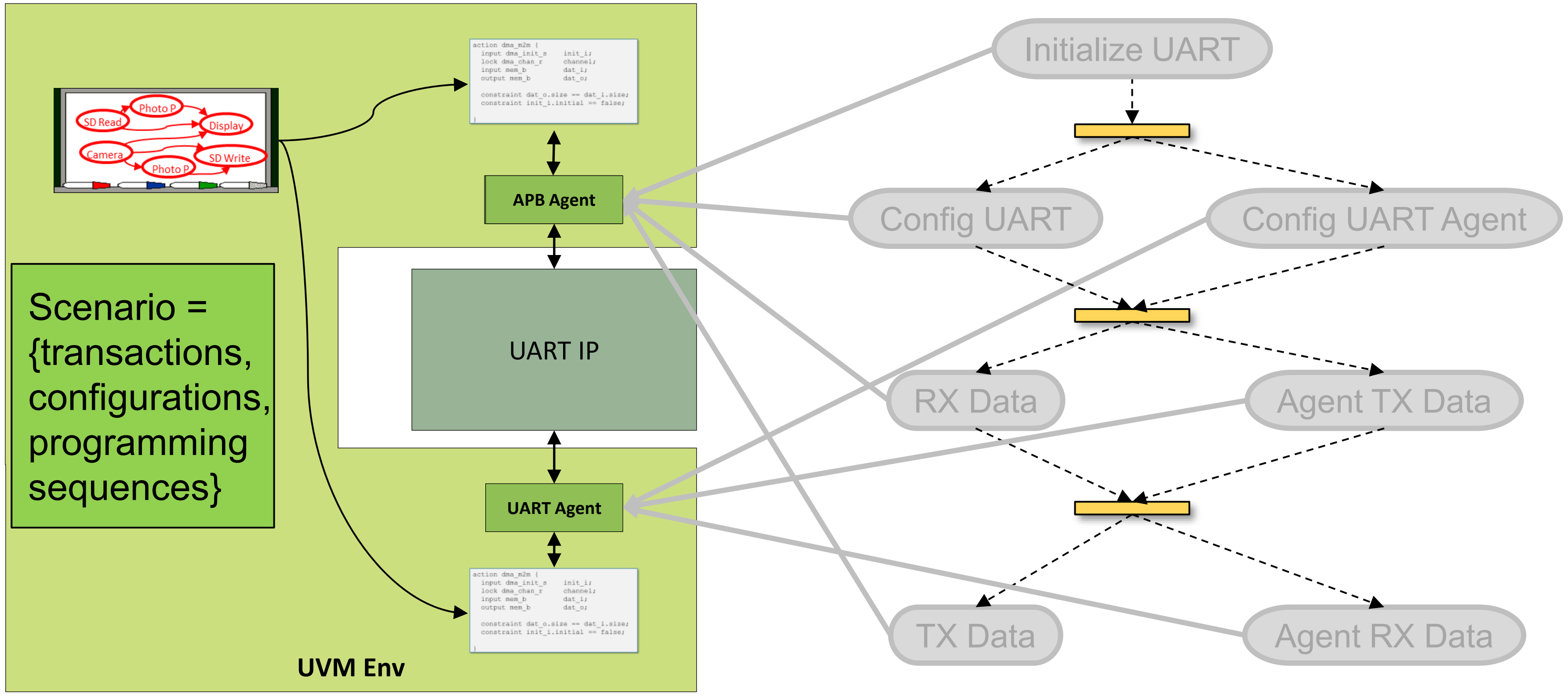
```
action dma_m2m {
  input dma_init_s    init_i;
  lock dma_chan_r     channel;
  input mem_b         dat_i;
  output mem_b        dat_o;

  constraint dat_o.size == dat_i.size;
  constraint init_i.initial == false;

}
```

```
action dma_m2m {
  input dma_init_s    init_i;
  lock dma_chan_r     channel;
  input mem_b         dat_i;
  output mem_b        dat_o;

  constraint dat_o.size == dat_i.size;
  constraint init_i.initial == false;

}
```

Concise Language to Specify Verification Intent

A *complement* to UVM, not a replacement

- Behavior = *Action*
- Schedule = *Activity*



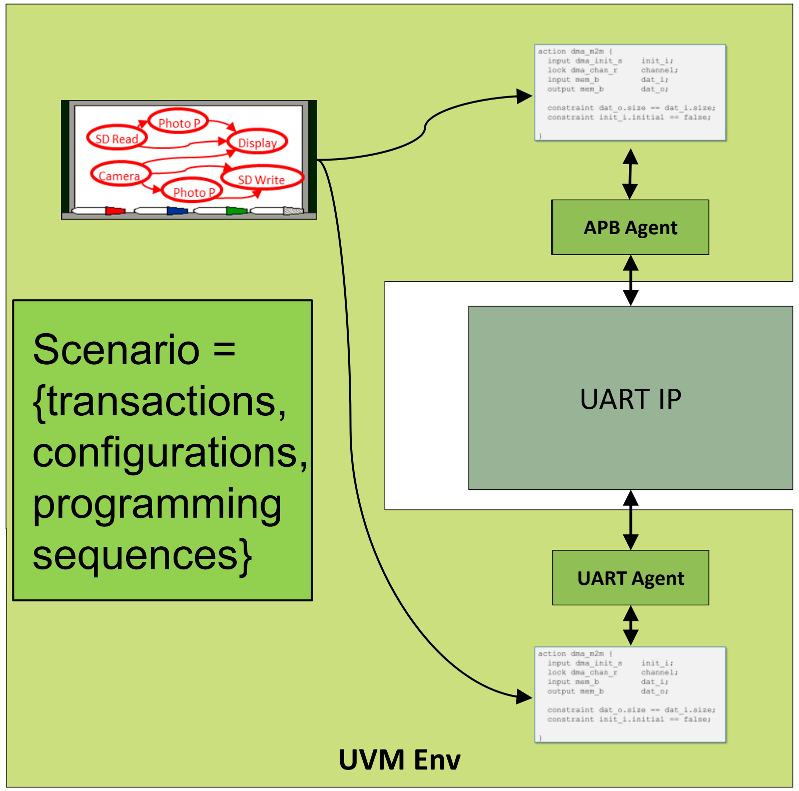
Scenario = {transactions, configurations, programming sequences}

```
action dma_m2m {
  input dma_init_s     init_i;
  lock dma_chan_r      channel;
  input mem_b          dat_i;
  output mem_b         dat_o;

  constraint dat_o.size == dat_i.size;
  constraint init_i.initial == false;

}
```

```
action dma_m2m {
  input dma_init_s     init_i;
  lock dma_chan_r      channel;
  input mem_b          dat_i;
  output mem_b         dat_o;

  constraint dat_o.size == dat_i.size;
  constraint init_i.initial == false;

}
```

Concise Language to Specify Verification Intent

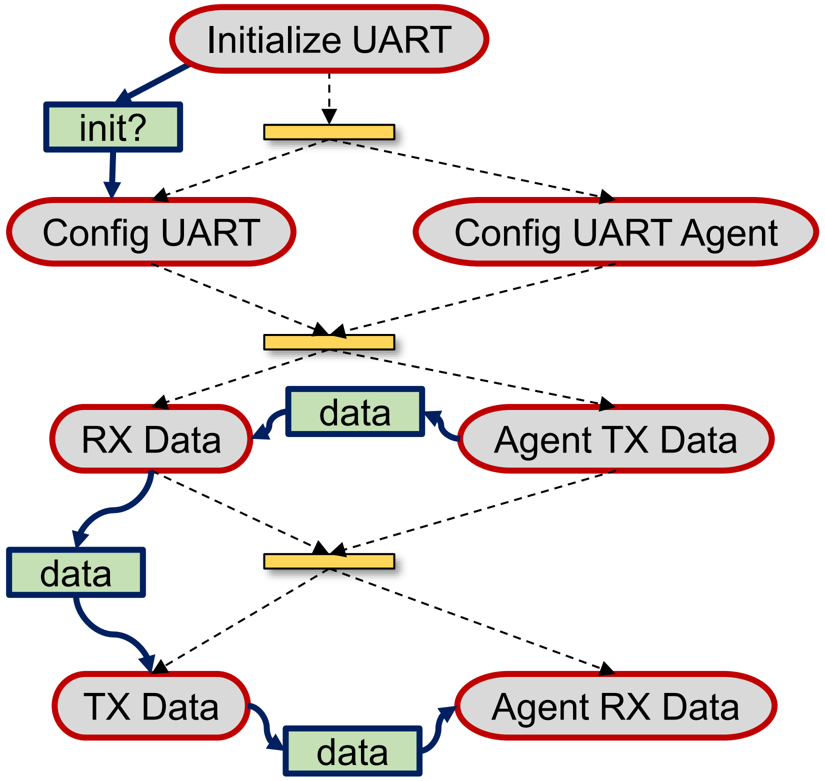
A *complement* to UVM, not a replacement



```
action dma_m2m {
  input dma_init_s     init_i;
  lock dma_chan_r      channel;
  input mem_b          dat_i;
  output mem_b         dat_o;

  constraint dat_o.size == dat_i.size;
  constraint init_i.initial == false;

}
```

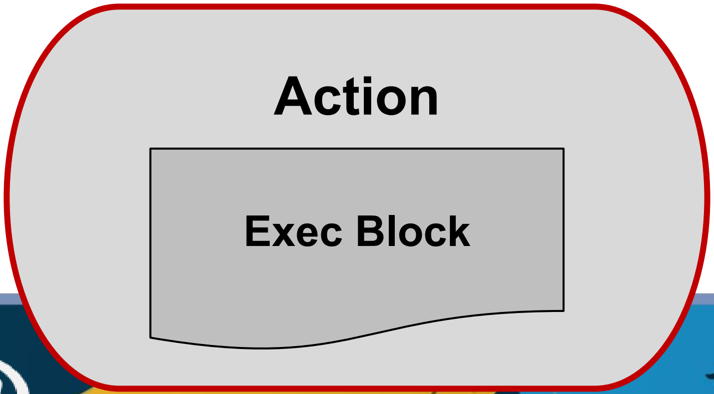
```
action dma_m2m {
  input dma_init_s     init_i;
  lock dma_chan_r      channel;
  input mem_b          dat_i;
  output mem_b         dat_o;

  constraint dat_o.size == dat_i.size;
  constraint init_i.initial == false;

}
```

Scenario = {transactions, configurations, programming sequences}



- Behavior = *Action*
- Schedule = *Activity*
- Sequential Data = *Buffer*
- Parallel Data = *Stream*
- State info = *State*
- In UVM, PSS can create a set of sequences
  - Run in existing UVM env

The Rubber Meets the Road



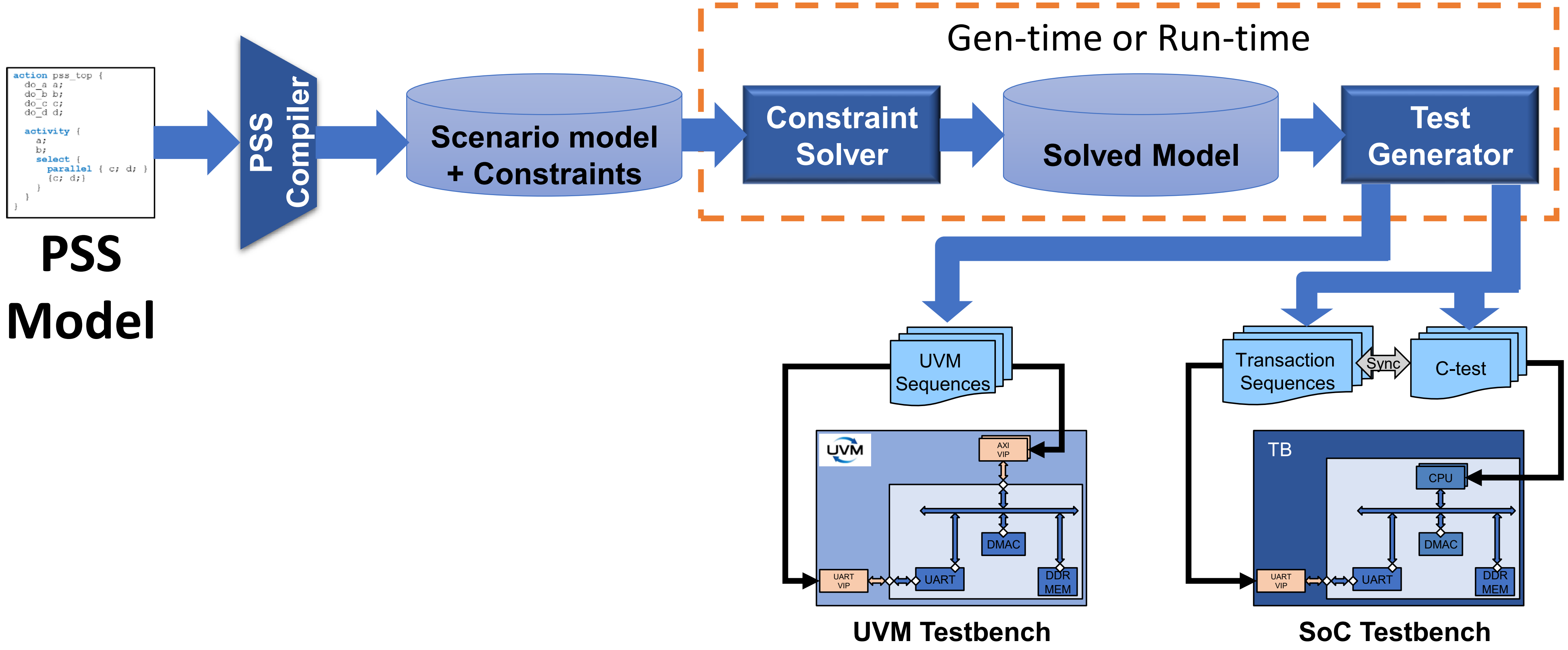
The Abstract Model must be implemented on different targets

Atomic Actions → target code

- Target code modeled in *exec* blocks

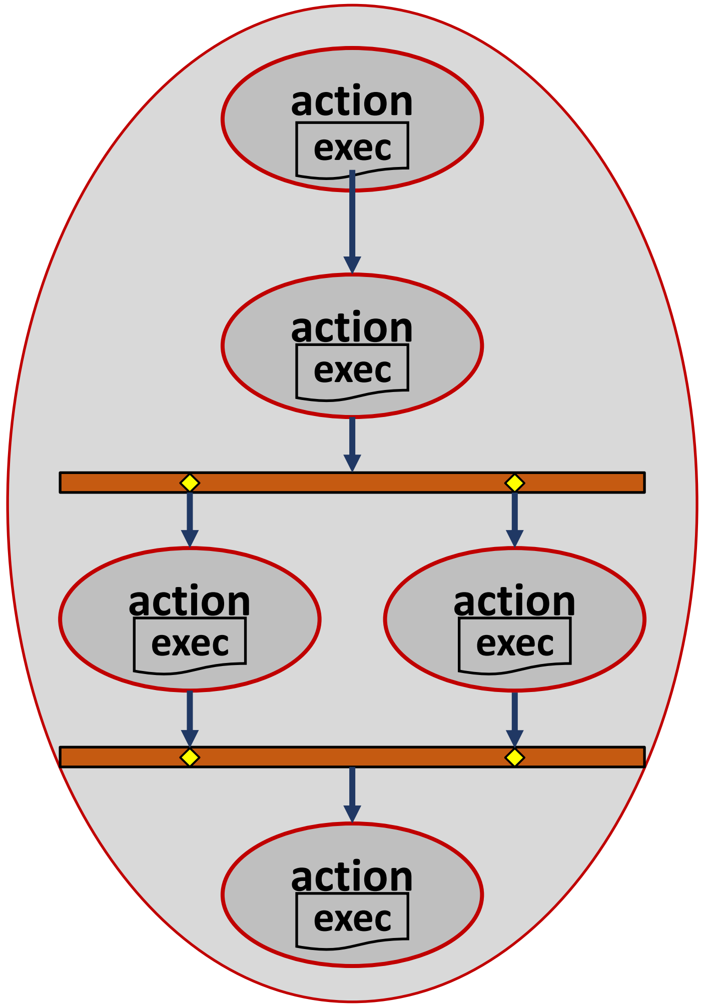
Generator assembles target code according to *Activity* schedule

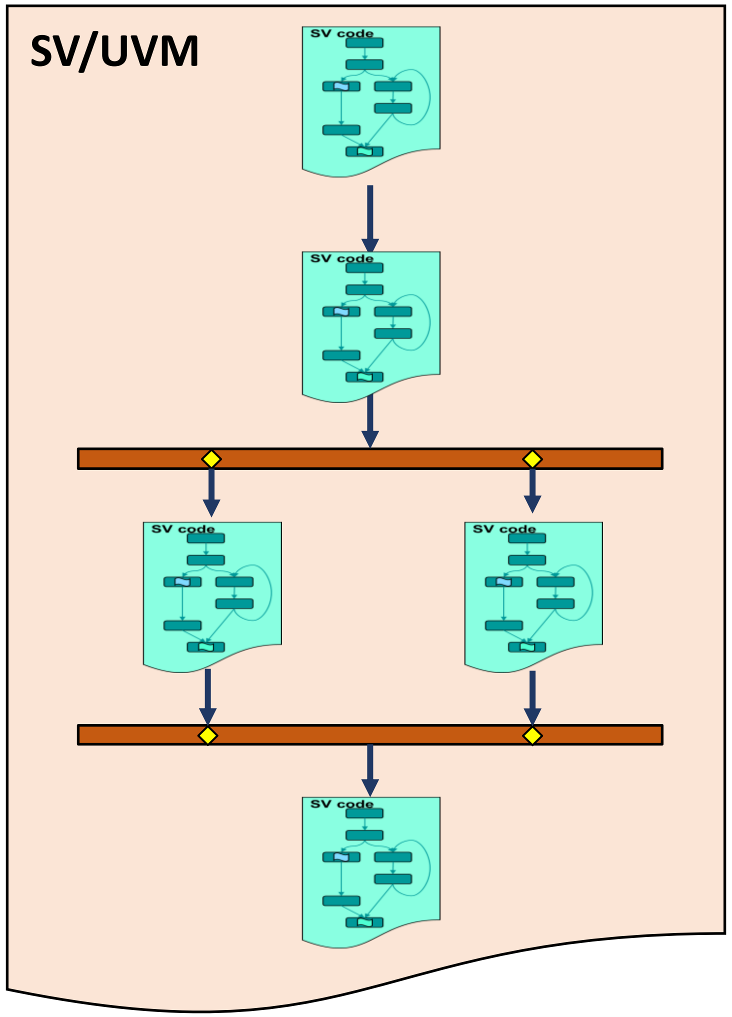
PSS Generalized Tool Flow

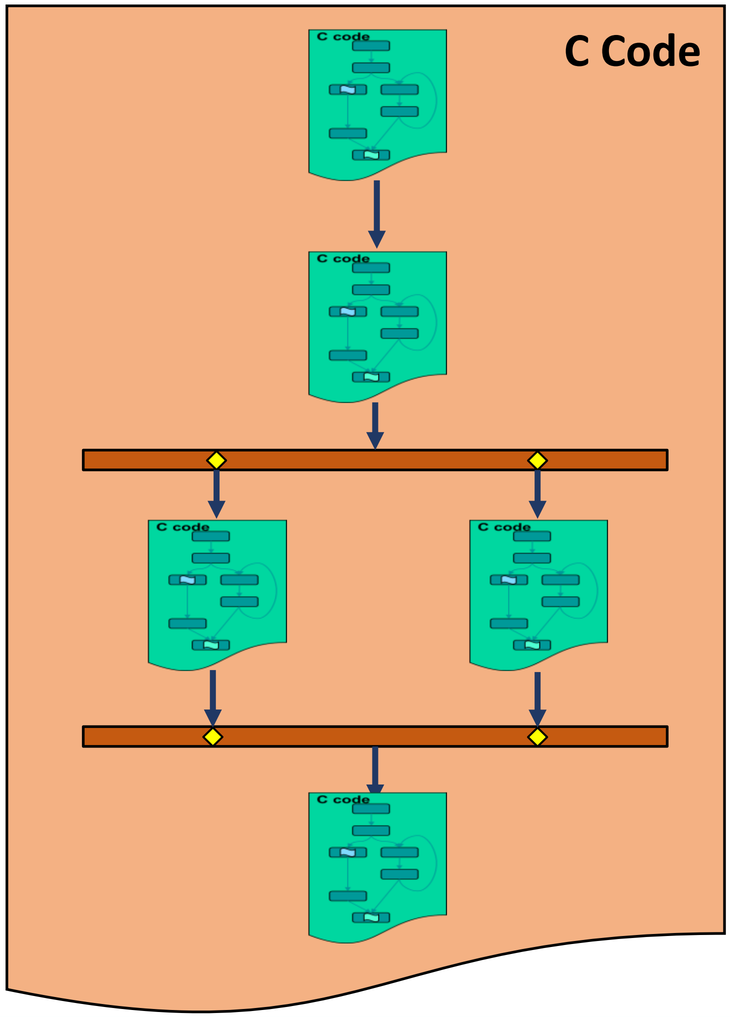


```
action pss_top {
  do_a a;
  do_b b;
  do_c c;
  do_d d;

  activity {
    a;
    b;
    select {
      parallel { c; d; }
      {c; d;}
    }
  }
}
```

Generated Code Implements Activity Schedule



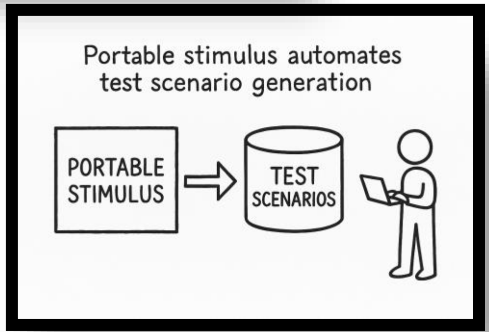


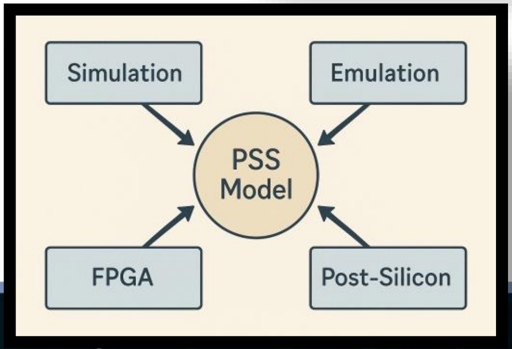


Effectively creating a robust PSS model

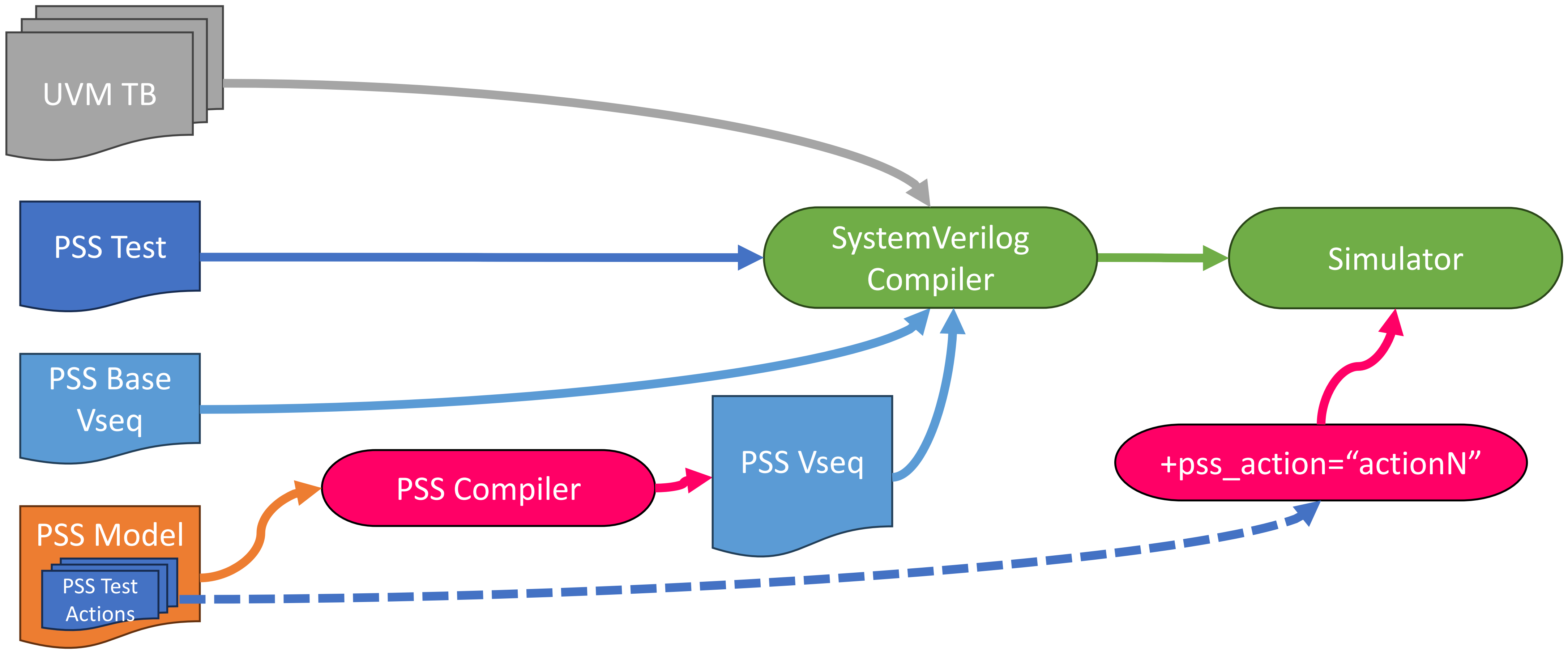
What is a PSS Model?

- A structured specification of **verification intent** written in the PSS language.
- Automating generation of **valid test scenarios.**
- **List of behavioural rules** on how to randomize and accommodate **tasks** and **variables** into scenarios.
- **Platform-independent** description that can be realized across multiple target environments



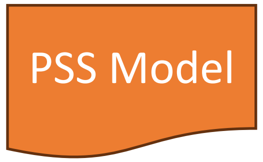


Proposed Flow (running PSS)



Starting a PSS Model

1. **Create a .pss file** to serve as the PSS Model entry point
2. **Organize model elements within package** declarations to enable reuse and organization
3. **Create a top component & action** that will serve as the entry point for scenario generation



```
package functions_pkg {


}
component ust_dma_c {
    import functions_pkg::*;














    action top {

      activity {





      }
    }
}
```

Splitting features into tasks

1. Group the system main functionalities into SV tasks in a Virtual Sequence.
2. Import the SV task into PSS.
3. Create actions in the PSS Model and connect them to the SV tasks.

```
task run_alloc (int manage_id);
  my_alloc_seq alloc_seq;
  alloc_seq = my_alloc_seq::type_id::create("alloc_seq");

  if (!alloc_seq.randomize() with {
    my_manage_id == local::manage_id;
  }) `uvm_fatal(report_id, "Unable to randomize alloc_seq.")

  alloc_seq.start(m_env.m_agt.seqr);

endtask : run_alloc
```

```
package functions_pkg {
  function void run_alloc(int manage_id);
  import target SV function run_alloc;
}
component ust_dma_c {
    import functions_pkg::*;

    action do_alloc {
        rand bit[32] manage_id;

        exec body {
          run_alloc(manage_id);
        };
    }

    action top {

      activity {



      }
    }
}
```

Setting up the constraints

- Pick variables that will be randomized at the PSS level (i.e., **PSS-controlled variables**).
- num_of_alloc_msgs shall be randomized from within the **1-3 range**.

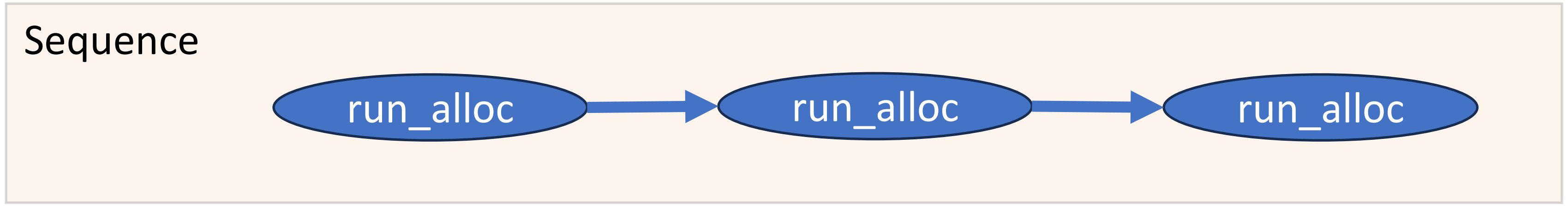
```
package functions_pkg {
 function void run_alloc(int manage_id);
 import target SV function run_alloc;
}
component ust_dma_c {
    import functions_pkg::*;

    action do_alloc {
        …
        exec body {
          run_alloc(manage_id);
        };
    }

    action top {


      action bit[3:0] in [1..3] num_of_alloc_msgs;

      activity {





      }
    }
}
```

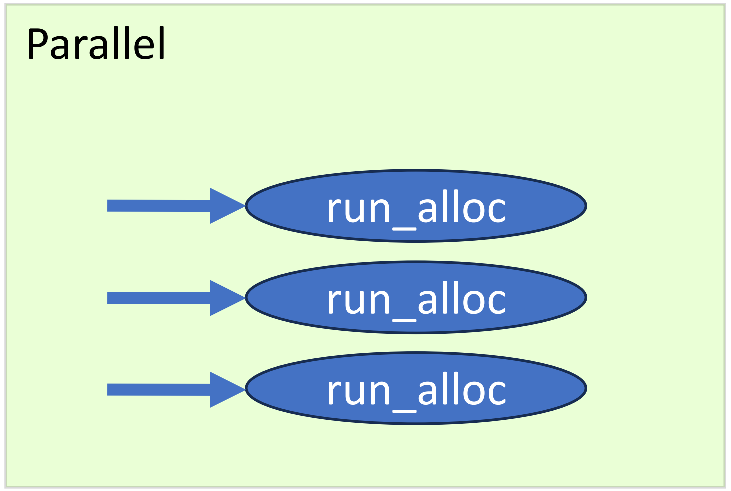
Calling actions

- Tasks shall be locally declared inside the top action.
- Start the activity block.
- Randomize the number of **alloc** messages.
- Run, in sequence, all alloc messages.

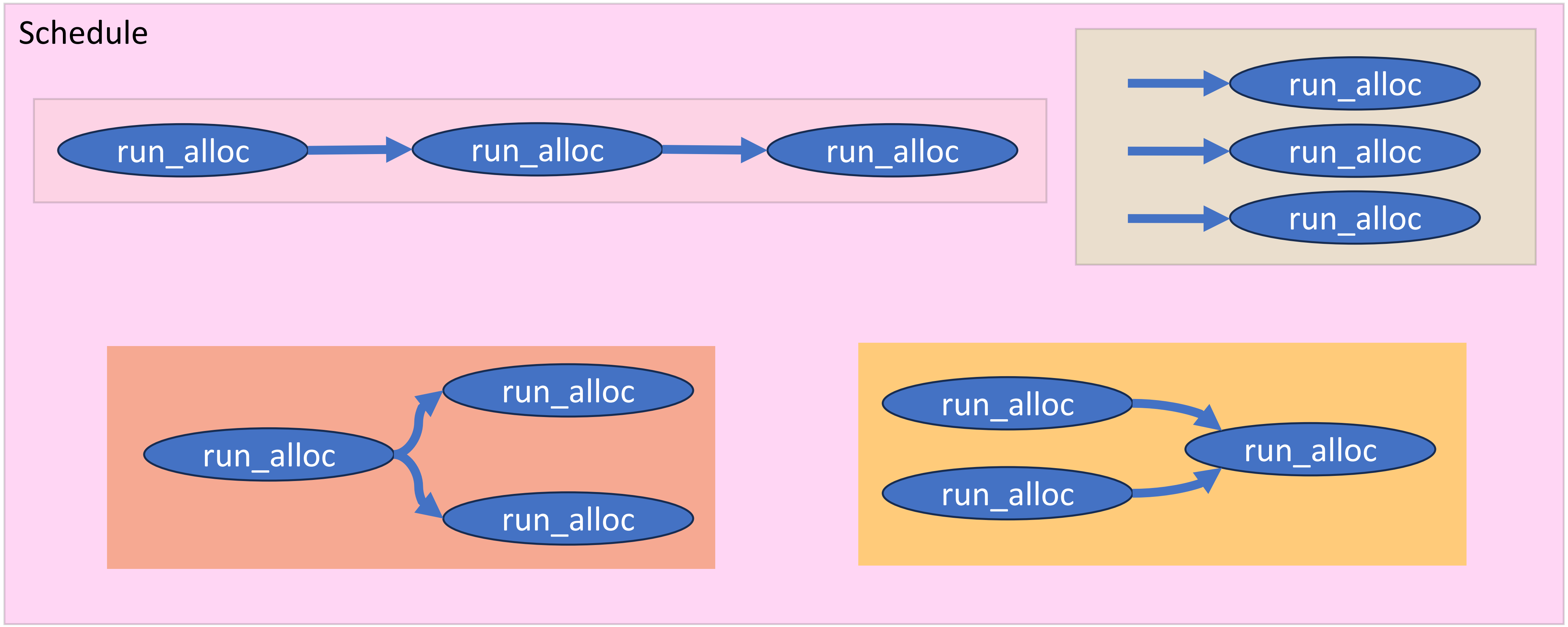
```
package functions_pkg {
 function void run_alloc(int manage_id);
 import target SV function run_alloc;
}
component ust_dma_c {
    import functions_pkg::*;

    action do_alloc {
        …
        exec body {
          run_alloc(manage_id);
        };
    }

    action top {
      do_alloc alloc;

      action bit[3:0] in [1..3] num_of_alloc_msgs;

      activity {
        num_of_alloc_msgs;
        sequence {
          do alloc; do alloc; do alloc;
        }
      }
    }
}
```

Activity: schedule, sequence, parallel



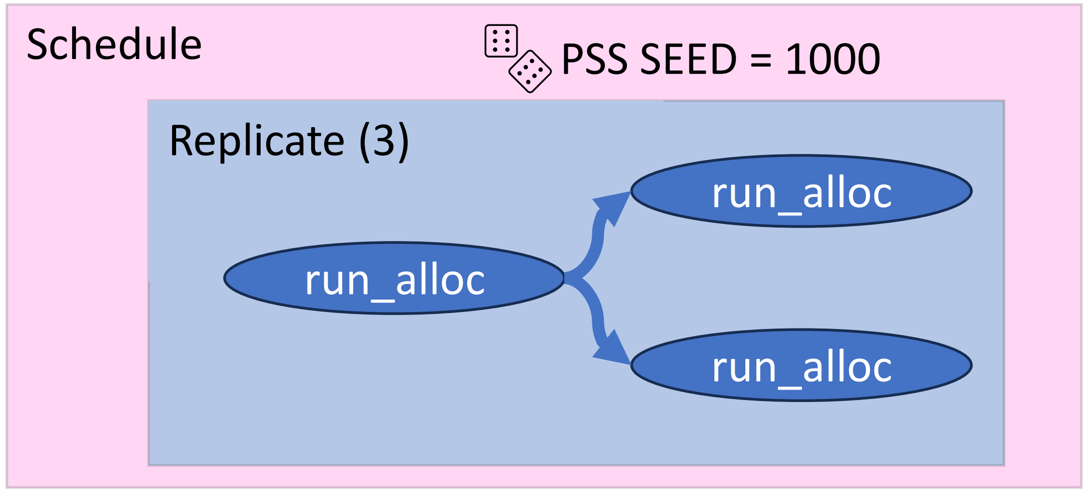


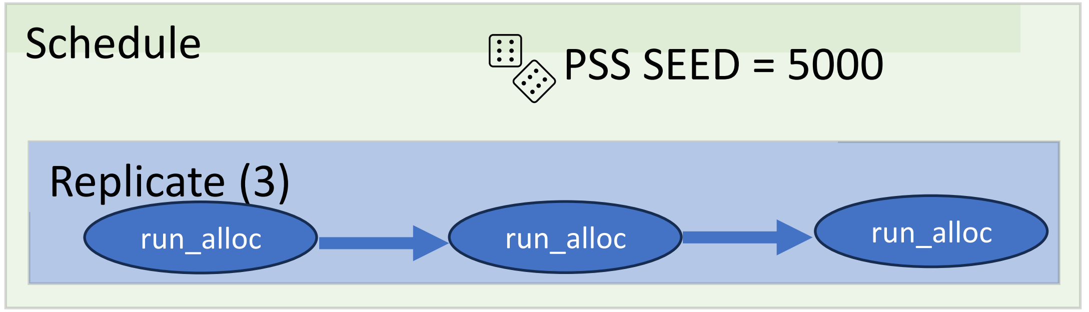
Activity: schedule, sequence, parallel



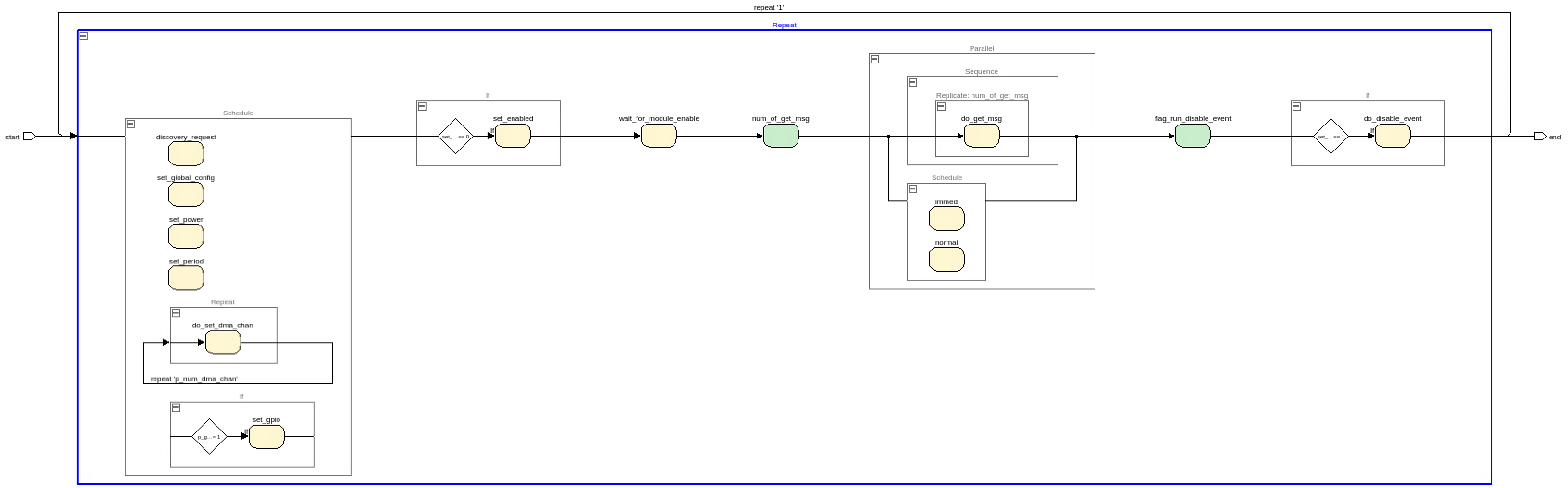
Debugging a PSS Model

- Enable trace logs to track action scheduling and execution flow
- Examine component hierarchy, action relationships, and scheduling in graphical views.
- Use fixed seeds to reproduce specific scenarios for detailed analysis.





Real-world Example



Summary

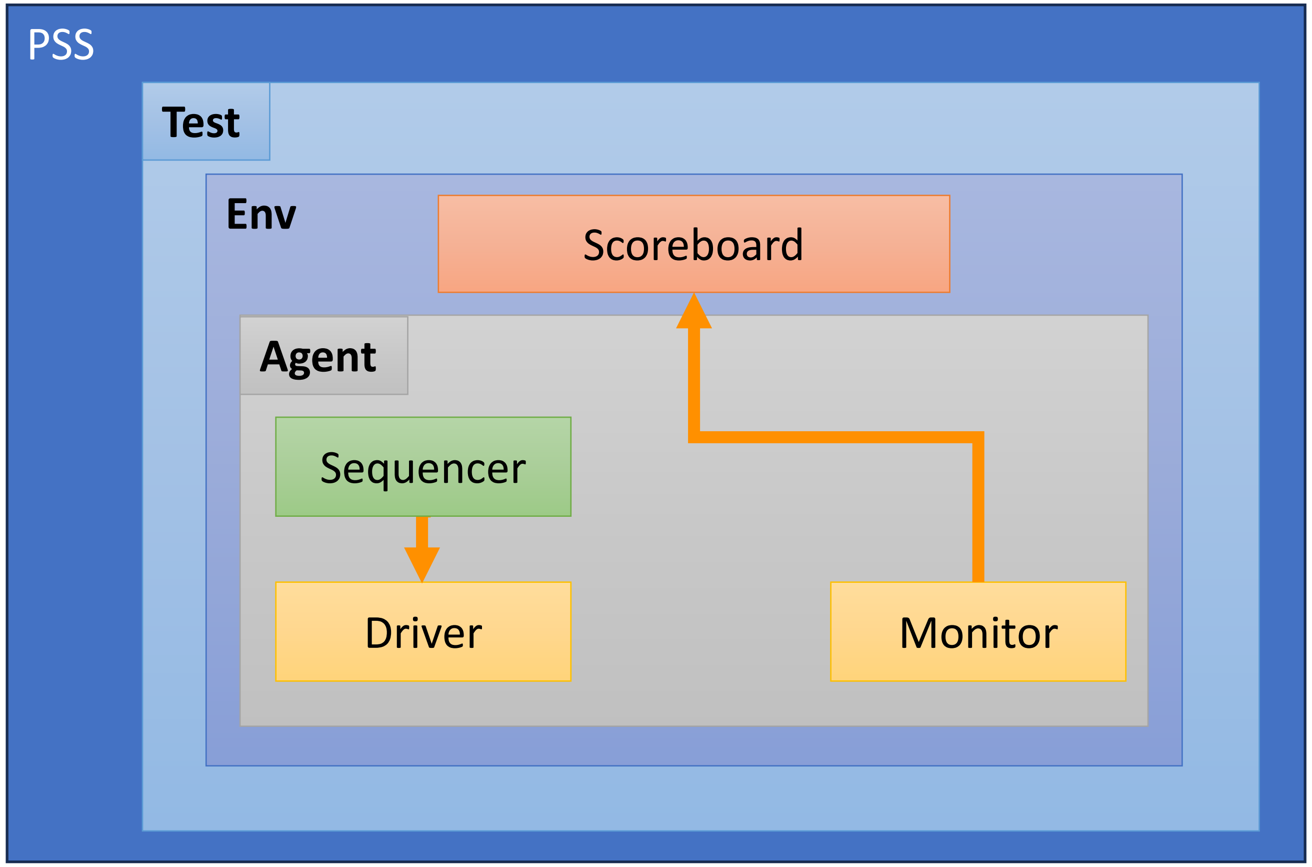
1. Split features into PSS tasks.
2. Clearly define value constraints for PSS-controlled variables.
3. Implement the PSS model.
4. Run any PSS visualization tool to see and graphically debug PSS scenarios.

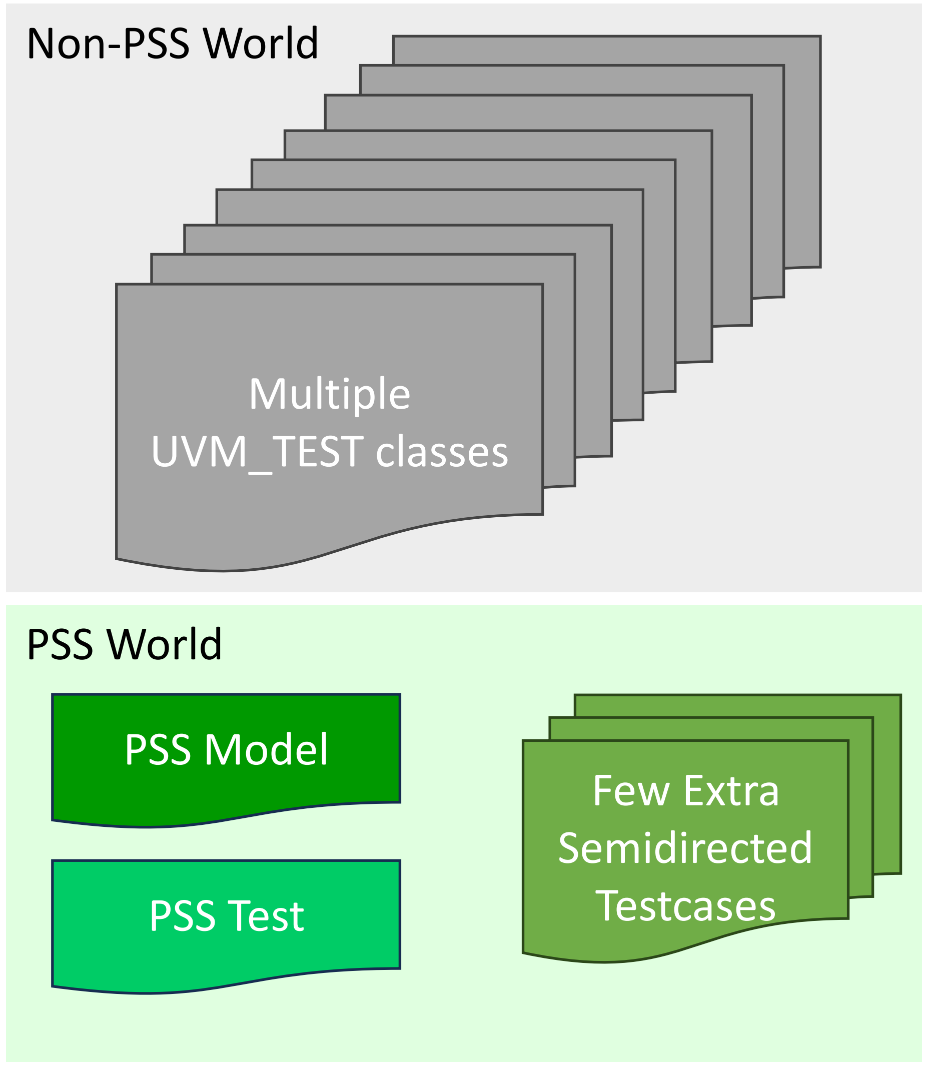
UVM & PSS

Introduction

- PSS aims at **enhancing** UVM testbenches.
- PSS **reuses** existing environments.
- PSS allows for **dynamic and randomized** testcase generation.

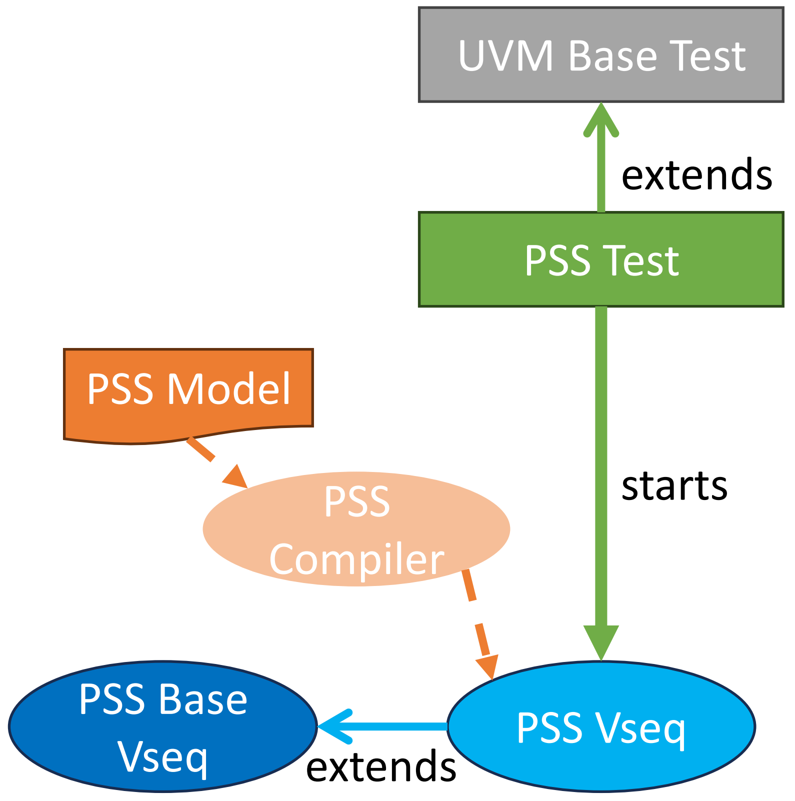
PSS **around** UVM





Basic Structure: **PSS Test**

- PSS reuses the **entire UVM testbench** (e.g., scoreboard, agents).
- The **PSS Test** extends a UVM_TEST to preserve the UVM structure.
- The **PSS Test** starts the compiled PSS vseq.
- *PSS Vseq is usually **unreadable**, but the implementation of the **PSS model;** it **extends the PSS Base Vseq**.*



Basic Structure: **PSS Test**

- The PSS Test's **run_phase** has the necessary code to start the PSS Vseq.
- The **PSS Vseq** shall be started as any other Vseq would.
- The **PSS Vseq** shall not target any sequencer/virtual sequence.

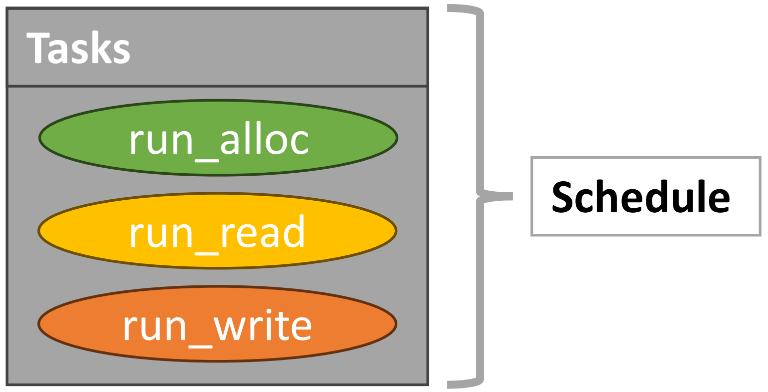
```
class dma_pss_test extends dma_base_test;
  …
  task run_phase_runSequences(uvm_phase phase);
      dma_qps_nvirt_vseq test_vseq;
      phase.raise_objection(this, "Starting Test");
      `uvm_info("TEST","PSS Test Running", UVM_HIGH);
      test_vseq = dma_qps_nvirt_vseq::type_id::create("pss_vseq");
      test_vseq.m_root_action = pss_root_action;
      test_vseq.start(null);
      phase.drop_objection(this, "Test Finished");
   endtask // run_phase
endclass : dma_pss_test
```

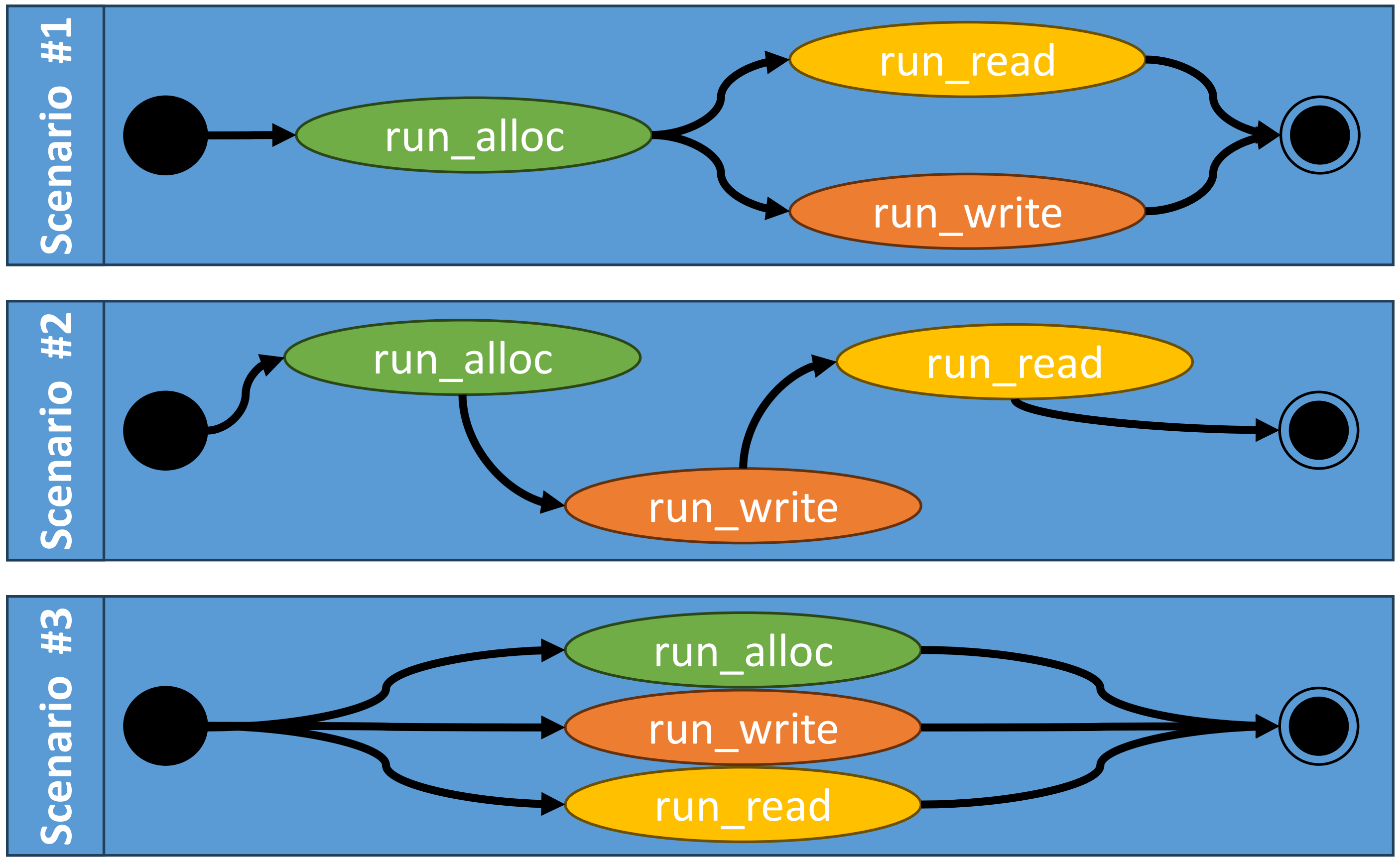
Basic Structure: **PSS Base Vseq**

- The **pss_base_vseq** is simply a **uvm_sequence** with the **list of available tasks**.
- Extends **uvm_sequence**.

```
class dma_pss_base_vseq extends uvm_sequence;
  …
  task run_alloc (int manage_id);
       my_alloc_seq alloc_seq;
       alloc_seq = my_alloc_seq::type_id::create("alloc_seq");

       if (!alloc_seq.randomize() with {
              my_manage_id == local::manage_id;
       }) `uvm_fatal(report_id, "Unable to randomize alloc_seq.")

       alloc_seq.start(m_env.m_agt.seqr);

  endtask : run_alloc

endclass : dma_pss_base_vseq
```

Example of Scenario Generation

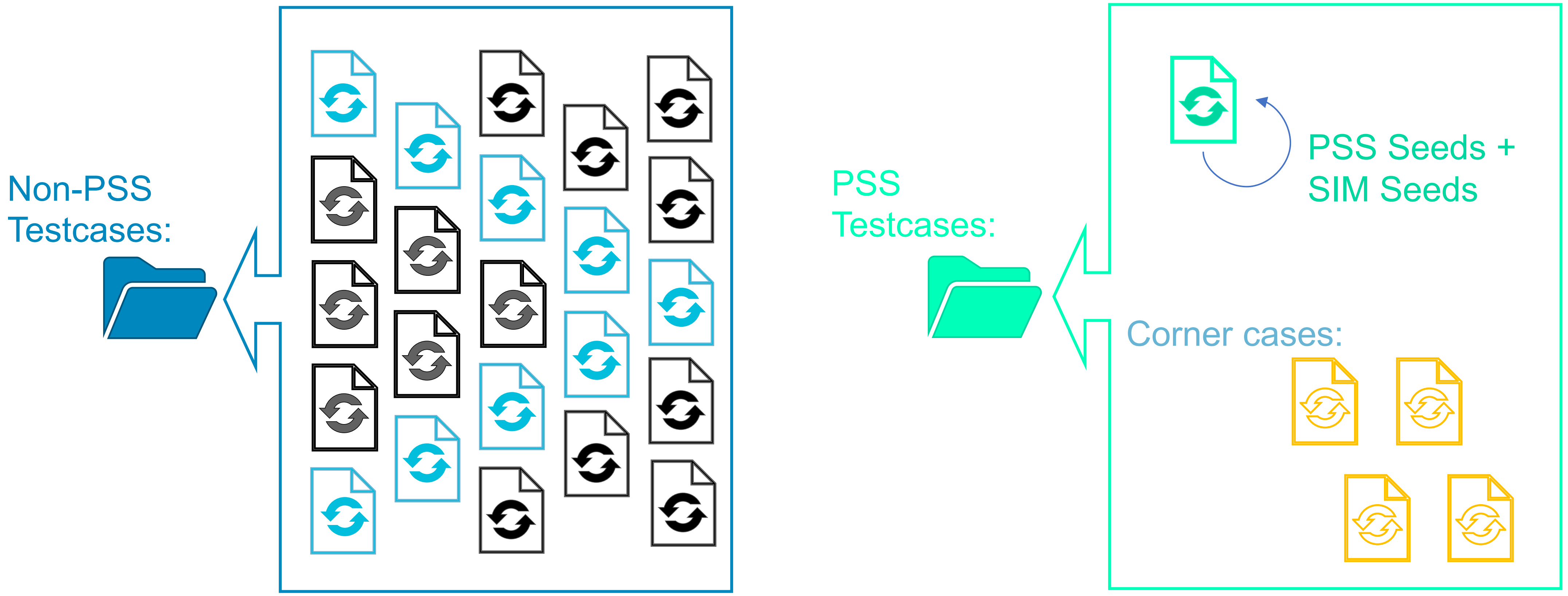




Scenario randomization is controlled by

PSS_SEED

PSS & Regressions



Considerations

- PSS manipulates the stimuli sent through a UVM environment.
- **PSS Test** starts a compiled PSS Vseq, which implements the PSS model.
- **PSS Base Vseq** shall contain a list of available tasks.
- **PSS_SEED** controls the generation of scenarios and allows for reproducing them.
- **PSS-based Regressions** focus on a PSS Test running with multiple PSS_SEEDs + few corner-case tests (if necessary).

Using PSS from day one

Introduction

- Using PSS from Day 1 is the best-case scenario.
- No need to adapt existing testcases/tasks into PSS.
- Liberty to decide the best approach to extract the maximum out of PSS without re-work.

Understand the Design

- Thoroughly understanding the design is the key!
- Think about ways to break RTL features into **tasks**.
- Implement a robust UVM environment and create all the necessary sequences and virtual sequences.
- Remember: **TASKS!**

Create the tasks

- Create the **PSS Base Vseq**.
- **PSS Base Vseq** shall extend uvm_sequence.
- Define the **major tasks** as SystemVerilog tasks in the **PSS Base Vseq**.
- If needed, create auxiliary helper tasks/functions.
  - These shall **not** be imported by the PSS model.

```
class dma_pss_base_vseq extends uvm_sequence;
  …

  task task_1 (args);
  task task_2 (args);
  …
  task task_N (args);


  task task_AUX (args);

endclass : dma_pss_base_vseq
```

Relationship between tasks & PSS Model

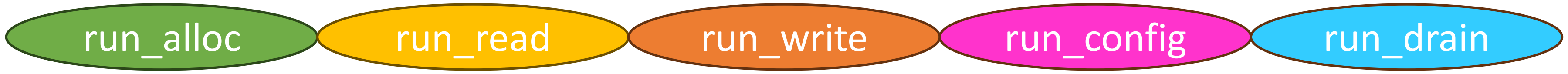
- Now let’s think about the relationship between tasks:
  - Which task(s) must **start** the simulation (e.g., **config**)?
  - Any other **hard requirement** for ordering tasks?
  - Which task must **finish** the simulation?

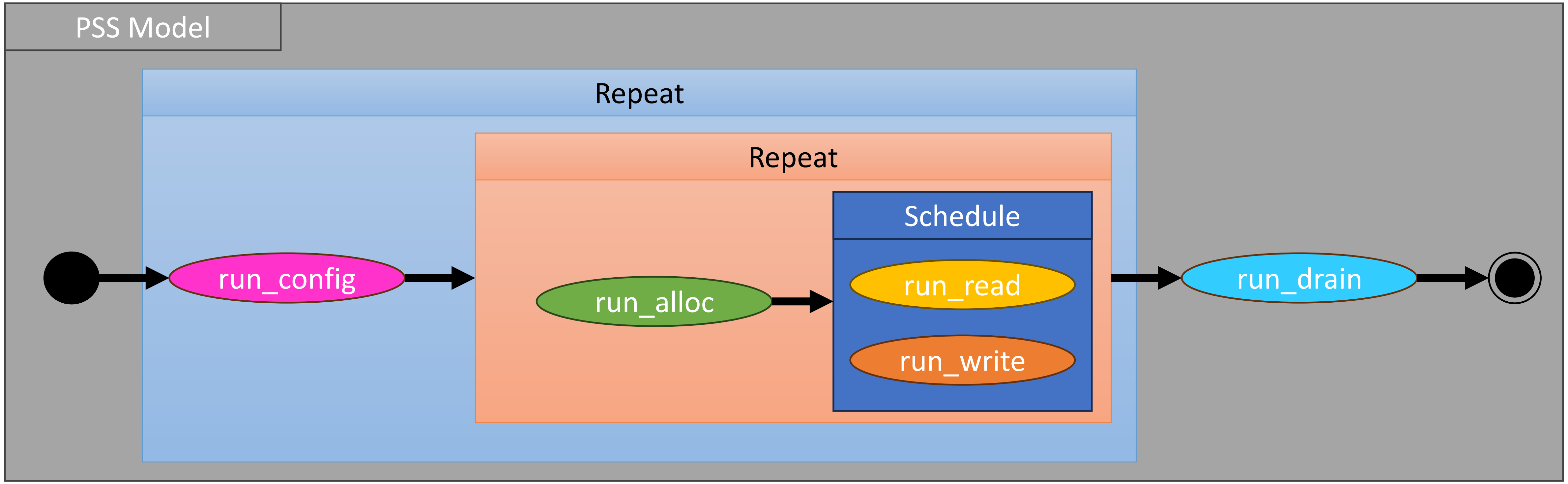
  - LEAVE SPACE FOR RANDOMIZATION! → *define tasks that can be executed in any order.*

- Implement tasks to allow randomization of variables at the PSS Model.

Relationship between tasks & PSS Model

- Example:





Create the PSS Test

- As previously presented, just create the PSS Test and start the compiled sequence from its **run_phase.**
- Ensure that it extends the UVM Base Test, which builds the environment.

```
class dma_pss_test extends dma_base_test;
  …

  task run_phase_runSequences(uvm_phase phase);
      dma_qps_nvirt_vseq test_vseq;

      phase.raise_objection(this, "Starting Test");

      `uvm_info("TEST","PSS Test Running", UVM_HIGH);
      test_vseq = dma_qps_nvirt_vseq::type_id::create("pss_vseq");

      test_vseq.m_root_action = pss_root_action;
      test_vseq.start(null);

      phase.drop_objection(this, "Test Finished");
   endtask // run_phase

endclass : dma_pss_test
```

Summary of Steps & Considerations

1. Understand the design spec.
2. Divide design features into different SV Tasks.
3. Define the relationship between tasks.
4. Implement the PSS Model
5. Create the PSS Test.

- PSS From Day 1 allows for designing a **PSS-compatible UVM environment**.
- Easier to define tasks.
- Possibility to re-use tasks in future non-PSS testcases.

Adapting existing UVM testbenches to support PSS

Assumptions

1. Have a working UVM testbench + set of tests
2. Want to use PSS to generate stimulus, instead of UVM tests
3. Want existing UVM tests to still be runnable, afterwards
4. Have set up a PSS flow that requires 3 main files to be defined, to run:
   - PSS base vseq
   - PSS model
   - PSS test

Objective: create these files and match/improve coverage achieved by current UVM tests.

Summary of PSS base vseq, model, and test

Example Filename	Description
my_pss_base_vseq.svh	Class definition which contains collection of tasks; each task is a ‘unit’ of stimulus
my_pss_model.pss	Imports tasks from **pss_base_vseq** and defines rules for how stimulus is generated from these tasks
my_pss_test.svh	Class definition of pss test; runs a vseq generated from **pss_model** + **pss_base_vseq**

- Location of each file will depend on how your PSS flow is set up
- Don’t forget to add these to any relevant packages and defns

Rough Outline of PSS Files

my_pss_base_vseq.svh

```
class my_pss_base_vseq
extends uvm_sequence;


  // declare handles for
     env, agents, configs;
     control knob vars

  // constrain vars

  // define tasks
     :
     :


endclass
```

my_pss_model.pss

```
package functions_pkg {
  // import tasks from
     base_vseq
};

component my_component {

  // define pss actions
     for imported tasks

  action top {

    activity {
      // define scheduling
         rules for actions
    };
  };

};
```

my_pss_test.svh

```
class my_pss_test
extends my_base_test;


  task run_phase(…);

    // create vseq
    // pass handles
    // start vseq

  endtask


endclass
```

Key Phases and Milestones

Basic PSS setup Compiles

- Ensure PSS vseq, test, and model correctly interact with existing environment.
- Ensure PSS flow works as expected.
- Minimal stimulus.

PSS Sanity Test Passes

- Create stimulus for basic sanity test of DUT, with PSS.
- Get familiar with PSS model syntax and capabilities – and your PSS tool's debug capabilities

PSS Model is Finalized

- Slowly incorporate more stimulus.
- Can 'decompose' key UVM tests into vseq tasks and ensure model can create the test cases as scenarios.
- Eventually scenarios cover all test cases.

Basic PSS Setup: Notes

- Vseq tasks will perform test-level functionality e.g.
  - Starting a seq on a sequencer that lives as *env.agent.sequencer*
  - Incrementing a transaction count variable
- So, vseq needs handles to UVM components, configs, and various variables
- These will likely live in the base test which the PSS test extends
- So can pass handles into vseq in run phase of PSS test, before starting vseq

```
task run_phase(uvm_phase phase);
  …
  // my_pss_vseq type is generated from model, extends my_pss_base_vseq
  vseq = my_pss_vseq::type_id::create("vseq");
  vseq.m_env = m_env;
  vseq.m_cfg = m_cfg;
  vseq.start(null);
  …
```

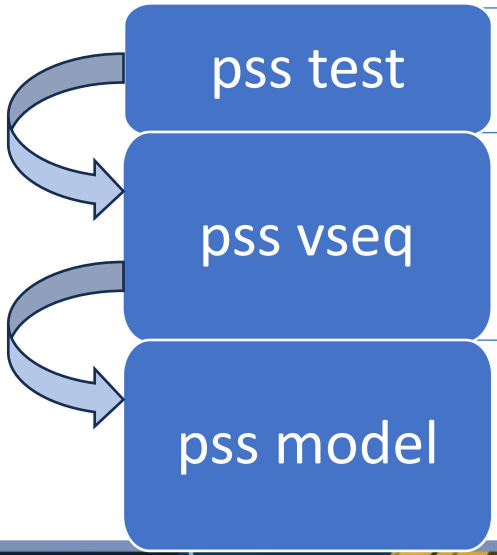
Common Pitfalls

⇒ **Problem:** One or more test cases requires overriding build phase.

⇒ **Solution:** Define *pss_test_<specialized>*; extends *pss_test*, overrides build phase.

⇒ **Problem:** Model or base_vseq needs vars that can't be directly passed to base_vseq.

⇒ **Solution:** Use a ‘pss_config’ object.



```
<pkg>::m_pss_cfg ={…}; //or: m_cfg.m_pss_cfg ={…}; vseq.m_cfg = m_cfg;
```

```
function void get_pss_cfg(output pss_cfg_t pss_cfg);
  pss_cfg = <pkg>::m_pss_cfg; //or: pss_cfg = m_cfg.m_pss_cfg;
endfunction
```

```
struct pss_cfg_t {…};
pss_cfg_t pss_cfg;
get_pss_cfg(pss_cfg);
```

Coverage Monitoring for PSS

- Coverage provides a metric for comparison against existing UVM tests
- Run PSS tests in separate regression
  - Instead of suites of different UVM tests, run same PSS test multiple times
  - Different PSS seeds between runs cover many scenarios per regression
  - Exception: *pss_test_<specialized>* with overridden build phase

- As PSS model gets finalized, PSS regression coverage rises to meet or exceed UVM-test coverage
- Again: might use a few semi-directed UVM tests for very edge cases

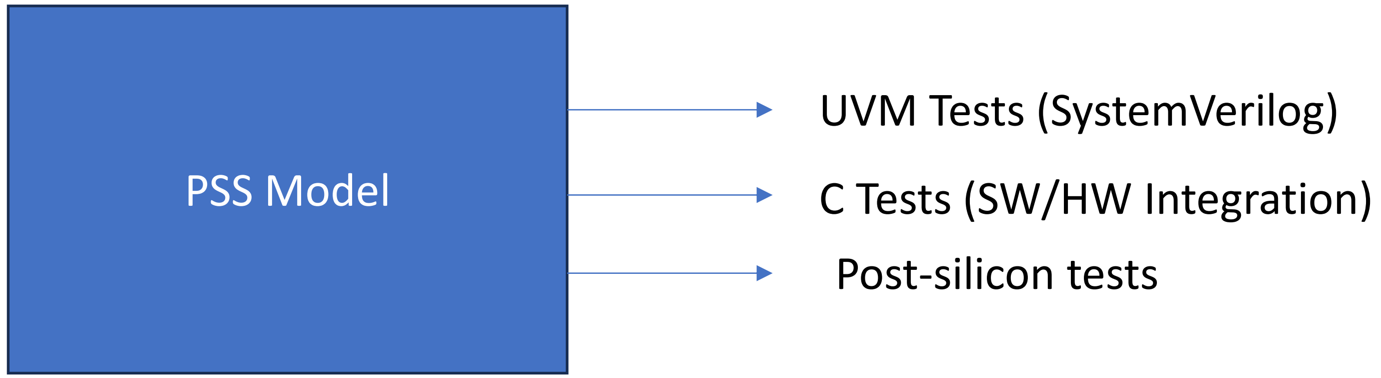
Portable Stimulus Standard, emphasizing PORTABLE

PSS Portability - The Core Advantage

- **True Portability**: PSS separates test intent from implementation details
- **Write Once, Run Anywhere**: Same abstract model drives verification across multiple:
  - Environments (simulation, emulation, FPGA)
  - Abstraction levels (block, subsystem, system)
  - Languages (SystemVerilog UVM, C/C++)
- **Verification Continuum**: Seamless transition from block-level to system-level testing

Multi-Target Test Generation

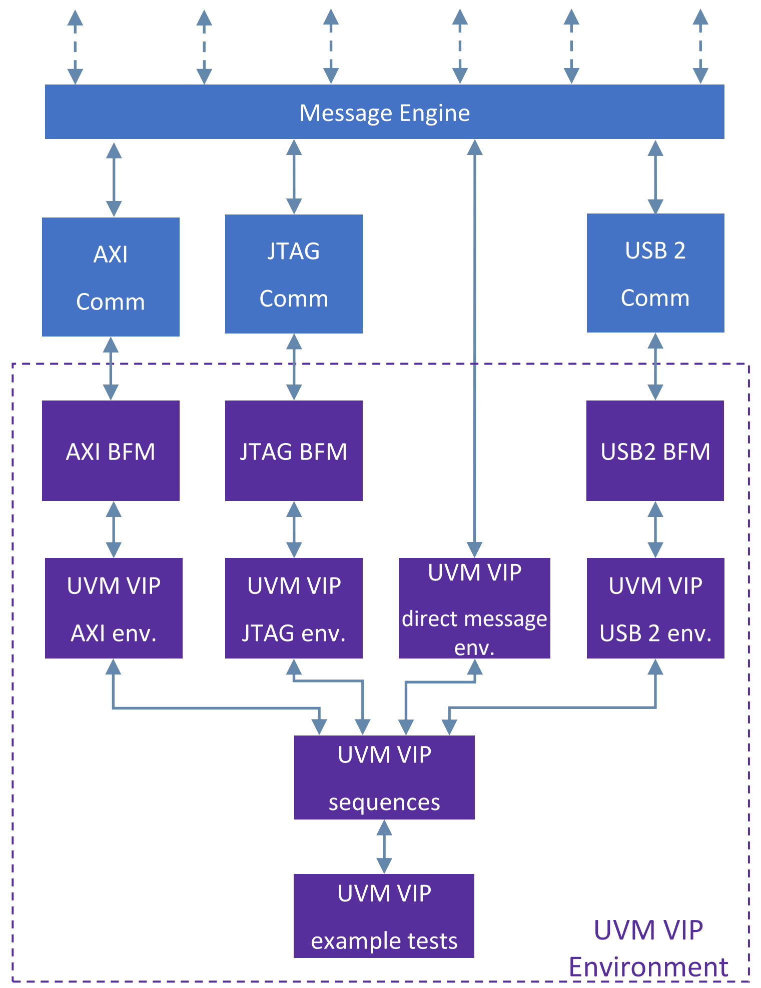
- One Model, Multiple Outputs



- **Consistent Test Intent**: Same scenarios used at multiple levels of abstractions / targets
- **Better Maintenance**: Tests definitions in one place, not multiple environments.

Scaling to Subsystem Level

- **From Block-Level to Subsystem**: Compose block level PSS models into subsystem scenarios
- **Reuse Without Rewrite**: Leverage existing block-level PSS components into larger context
- **Cross-Block Interactions**: Model complex behaviors spanning multiple design units

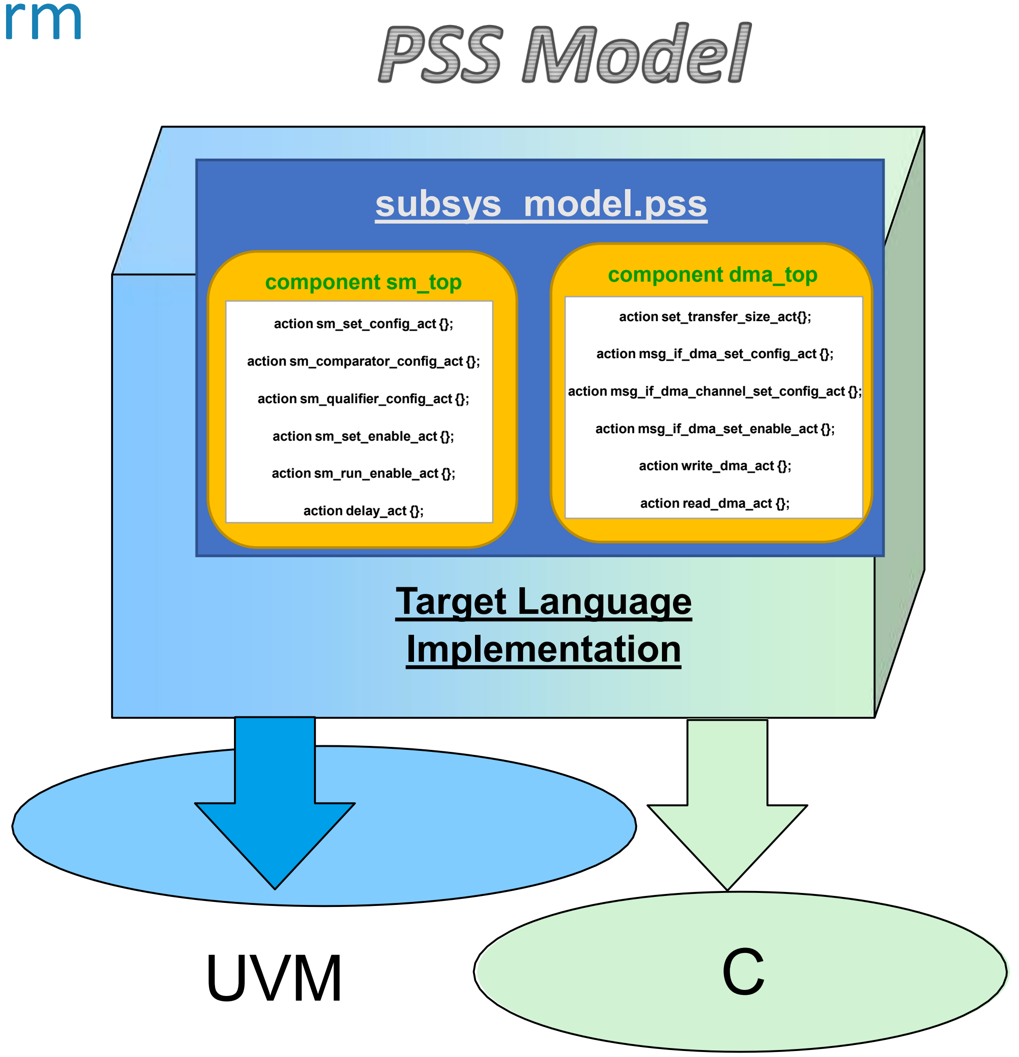


Scaling to Emulation and FPGA

- **Performance with Portability**: Run the same tests on acceleration platforms
- **Test Reuse Strategy**:
  - Simulation → Debug detailed behaviours
  - Emulation → Run longer, more complex tests
  - FPGA → Test in near-real-time environment
- **Implementation Considerations**: What changes between environments vs. what stays the same

Implementation Example: Tessent Embedded Analytics subsystem cross-platform Test Generation

- Added PSS model for test / sequence generation for.
  - **Unified Testing Framework** — Generate identical test scenarios for both UVM and C-based environments
  - **Seamless Environment Transitions** — Maintain test consistency across development phases
- **Key Benefit:** Software teams can begin validation earlier in the development cycle, reducing time-to-market and improving quality through consistent testing methodology.

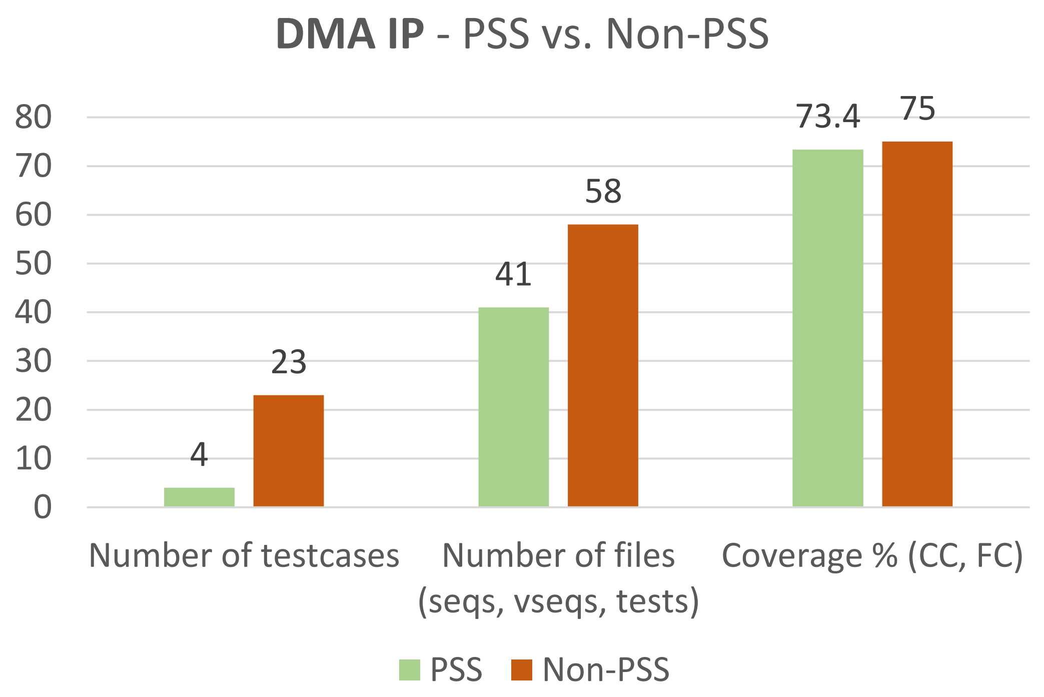

PSS Model
subsys_model.pss
component sm_top
action sm_set_config_act {};
action sm_comparator_config_act {};
action sm_qualifier_config_act {};
action sm_set_enable_act {};
action sm_run_enable_act {};
action delay_act {};
component dma_top
action set_transfer_size_act{};
action msg_if_dma_set_config_act {};
action msg_if_dma_channel_set_config_act {};
action msg_if_dma_set_enable_act {};
action write_dma_act {};
action read_dma_act {};
Target Language Implementation
UVM
C

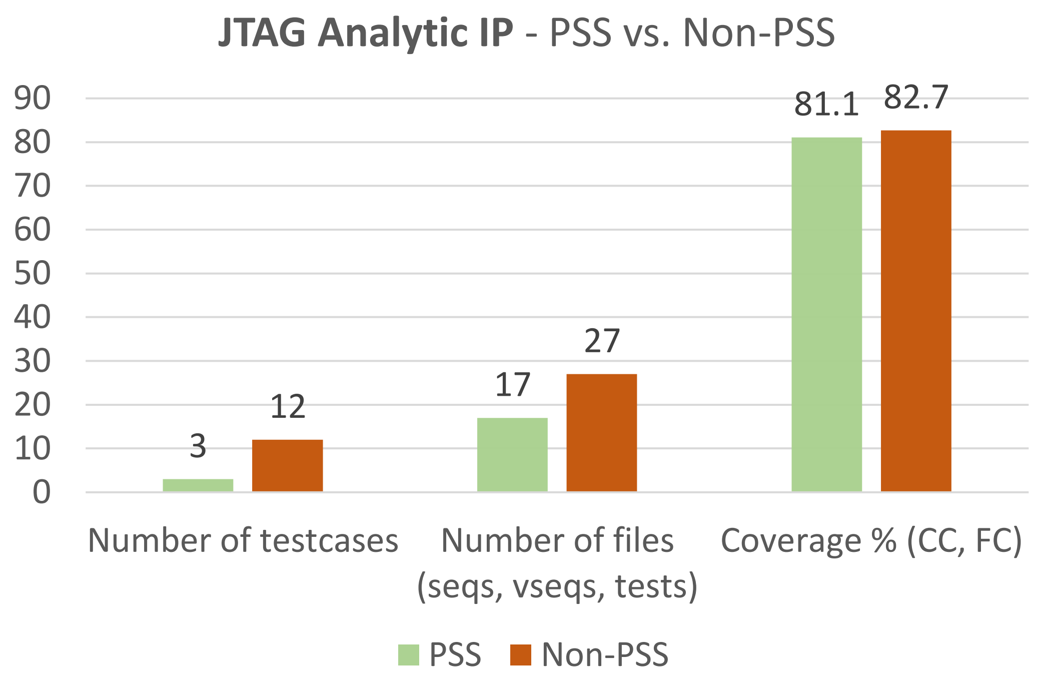
PSS Benefits and ROI

- **Verification Efficiency**: Write once, verify everywhere
- **Reduced Maintenance**: Single source of truth for test intent
- **Accelerated Coverage**: Reach complex scenarios earlier in the process
- **Smoother Hardware Transition**: Same tests from simulation to silicon

Real-world metrics on the impact of PSS

PSS vs. Non-PSS – tests and coverage





→ **4-5x reduction** in number of testcases to achieve the same coverage levels

→ **Many hours of development time saved!**

Advantages of PSS

1. **Reduced complexity + improved maintainability:**
   a. Much fewer files to manage.
   b. All platform-specific implementation centralized in a **single place** (i.e., *pss_base_vseq*)
   c. All higher-level verification intent centralized in a **single place** (i.e., PSS model)
2. **Dynamic and automatic** generation of scenarios:
   a. Random scheduling process – can be constrained as required
   b. Random variables and values – can be constrained as required
   → Helps **expose RTL bugs** as well as **testbench weaknesses**
3. Still reaches **similar levels of coverage** when compared to non-PSS implementations.

Conclusion

- PSS enhances the DV process by creating random scenarios.
- PSS is based on a ‘PSS Model’ which captures verification intent.
- PSS works with UVM – complementary methodologies.
- New project using PSS from day one facilitates efficient implementation.
- Existing UVM Environments can be adapted to work with PSS.
- PSS tests can work in other platforms (e.g., C and post-silicon).

Questions?