# Taming Configuration Complexity: A UVM-Based Approach To IP Verification

Bhaskar Vedula, Stephen Haake, Chandrakanth Betageri, Ganesh Sharma

Intel Corporation

2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

## INTRODUCTION

**Key Highlights & Problem Statement:**

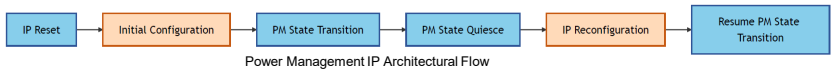
- Modern SoCs use highly configurable IPs for flexibility and reuse.
- Dynamic modification of IP behavior at reset and runtime is essential for performance/power, but creates verification challenges (many registers, modes, state transitions).
- Intel legacy static testbenches can't handle dynamic reconfiguration or scenario customization, leading to code duplication and maintenance overhead.

**Benefits:**

- Reduces development effort.
- Enhances reuse and improves coverage.
- Supports dynamic state transitions.

**What This Paper Proposes:**

- A modular, scalable UVM-based testbench methodology.
- Unified random configurable sequence pattern.
- Hierarchical sequence abstraction.
- Factory override mechanisms using native UVM.
- Case study: Highly configurable Power Management (PM) IP dramatically increases verification complexity because it must manage dynamic power-state transitions while supporting timing-sensitive reconfiguration, creating a huge space of register-programming permutations to validate. The figure below shows the architectural flow for the PM IP state transition.




Power Management IP Architectural Flow

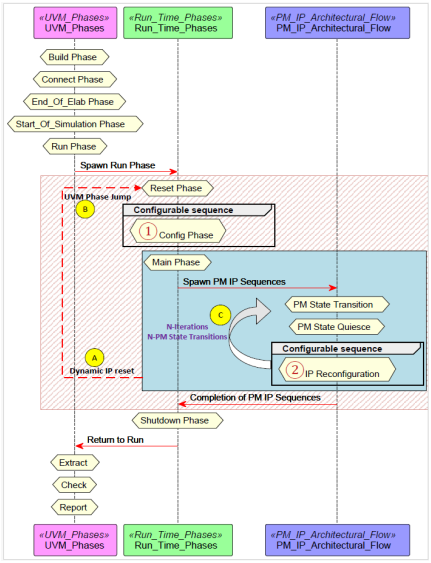
## UVM Phasing & Power Management IP Flow

**UVM Phasing & PM IP Flow:**

- PM IP dynamically modifies configurations to manage SoC power states.
- Register updates: Some anytime, some only when PM state transitions are paused, some require a reset.
- Uses UVM Run Phase and custom phases for coordinated, flexible execution.
- UVM Jump Phase enables asynchronous reset handling.

**Key Implementation:**

- Single unified configurable sequence for both initial and runtime configuration.
- Sequence configuration object enables scenario-specific control.
- Hierarchical sequence pattern maximizes reuse and scalability.




## Configurable Testbench Setup

**Unified Configurable Sequence:**

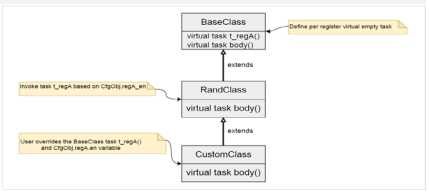
- A unified random configurable sequence is controlled through a sequence configuration object.
- One reusable sequence drives both initial configuration and runtime reconfiguration.

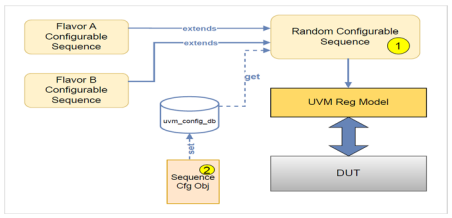
**Script-generated Sequence Configuration Object :**

- **Global knobs:** Skip config, reconfigure all.
- **Per-register controls:** Enable bits, randomization controls.
- **Default constraints:** Define architectural defaults and allow clean overrides.

**Environment Configuration Object:**

- Holds a shared handle to the sequence configuration object.
- Ensures all sequences/layers access a consistent configuration and register-model context.
- Provides synchronized control of configuration behavior across the entire testbench.




```
class seq_object extends uvm_object;

  // global controls
  rand bit none; // when 1, completely skip configuration
  rand bit reconfigure_all; // when 1, force all legal registers to reconfigure

  // per register enable and randomize controls
  rand bit regA_en, regA_rand_en;
  rand bit error_severity_control_en, error_severity_control_rand_en ;
  rand bit idle_power_managment_policy_en, idle_power_managment_policy_rand_en ;
  // skip registers used for functional flows we don't want to touch
  // rand bit dvfs_control_en, dvfs_control_rand_en ;
  // skip read-only registers that cannot be configured
  // rand bit current_device_state_en, current_device_state_rand_en ;

  `uvm_object_utils_begin(seq_object)
    `uvm_field_int(regA_en, UVM_DEFAULT)
    `uvm_field_int(regA_rand_en, UVM_DEFAULT)
    `uvm_field_int(error_severity_control_en, UVM_DEFAULT)
    ...
  `uvm_object_utils_end

  //Default constraints
  constraint c_reg_en {
    soft regA_en == 0;
    soft regA_rand_en == 1;
    // can only be configured on reset, reset sequence will override enable to 1
    soft error_severity_control_en == 0;
    soft error_severity_control_rand_en == 0;
    // configurable at reset and when PM flows are paused, reconfigure by default
    soft idle_power_managment_policy_en == 0;
    soft idle_power_managment_policy_rand_en == 1;
    ...
    }
  ...
  ...
endclass : seq_object
```

## Layering Of Sequence

**Hierarchical Sequence Pattern:**

**1. Base Layer**

- Defines per-register virtual tasks.
- Manages access to register model + environment configuration.
- Provides virtual tasks/functions for each register operation and default randomization.

**2. Random Layer**

- Executes base class tasks based on the sequence config object (enable / rand_en knobs).
- Supports random and constrained random flows.
- Drives bulk register programming for initial and runtime reconfigurations.

**3. Custom Layer**

- Designed for scenario-specific behavior.
- Users override base tasks for non-random or specialized flows.
- Enables architectural states via UVM type overrides (no code modification).
- Usage of Command-line override: +uvm_set_type_override=RandClass,CustomClass

```
class BaseClass extends uvm_sequence;
  `uvm_object_utils(BaseClass)
  virtual task body();
    if (!uvm_config_db #(my_config)::get(null, "*spr*", "cfg", cfg)) begin
      `uvm_fatal(get_name(), "DUT must get the config handle");
    end
    // empty shell
  endtask: body
  ...
  virtual task t_regA();
    //This is an empty virtual task.
  endtask: t_regA
  ...
  virtual task t_regARand();
    cfg.regblock.regA.randomize();
    cfg.regblock.regA.update();
    `uvm_info(get_name(), "regA is randomized", UVM_LOW);
  endtask: t_regARand

  virtual task t_error_severity_control();
    //This is an empty virtual task by default. User overrides in the custom layer
  endtask: t_error_severity_control

  virtual task t_error_severity_controlRand();
    cfg.regblock.error_severity_control.randomize();
    cfg.regblock.error_severity_control.update();
    `uvm_info(get_name(), "error_severity_control is randomized", UVM_LOW);
  endtask: t_error_severity_controlRand
  ...
endclass: BaseClass
```

```
class RandClass extends BaseClass;
  `uvm_object_utils(RandClass)
  virtual task body();
    super.body();
    if (cfg.cfgSeq.none == 1'b1) begin
      `uvm_info(get_name(), "Bypassing configure sequence", UVM_LOW);
    end else begin
      // call BaseClass tasks based on config variables
      // default register.field.set() calls
      if ((cfg.cfgSeq.reconfigure_all == 1) || (cfg.cfgSeq.regA_en == 0))
        t_regA();
      if ((cfg.cfgSeq.reconfigure_all == 1) || (cfg.cfgSeq.error_severity_control_en ==
1))
        t_error_severity_control();
      // ...
      // register.field.randomize() calls
      if ((cfg.cfgSeq.reconfigure_all == 1) || (cfg.cfgSeq.regA_rand_en == 1))
        t_regARand();
      if ((cfg.cfgSeq.reconfigure_all == 1) ||
(cfg.cfgSeq.error_severity_control_rand_en == 1))
        t_error_severity_controlRand();
      // ...
    end
  endtask: body
  // ...
endclass: RandClass
```

```
class CustomClass extends RandClass;
  `uvm_object_utils(CustomClass)
  virtual task body();
    // disable BaseClass randomization and test a specific policy configuration
    cfg.cfgSeq.idle_power_managment_policy_en = 1;
    cfg.cfgSeq.idle_power_managment_policy_rand_en = 0;
    ...
    super.body();
  endtask: body
  ...
  // override BaseClass with the desired fixed values
  task t_idle_power_managment_policy();
    cfg.regblock.idle_power_managment_policy.fieldX.set(0);
    cfg.regblock.idle_power_managment_policy.fieldY.set(1);
    cfg.regblock.idle_power_managment_policy.update();
  endtask
  ...
endclass: CustomClass
```

## IP Reconfiguration

**IP Reconfiguration trigger:**

- After applying PM stimuli, the flow executes PM State Quiesce to complete outstanding transitions and inhibit new ones before reprogramming.
- Reconfiguration occurs when PM IP enters a quiescent state.

**Reuse principle:**

- During IP reconfiguration, the methodology reuses the same IP Config Wrapper Sequence (same configurable sequence framework) rather than building a new sequence.

**Runtime mechanism:**

- A reconfig sequence waits for quiesce, optionally tweaks cfg knobs (example shown disabling a register's randomization), then invokes the wrapper sequence.
- Supports multiple reconfiguration iterations in a single simulation.

**End-To-End main flow:**

- PM transitions → quiesce + reconfig → resume PM transitions (all within the run-time phasing approach).

```
class ipCfgSeq extends uvm_sequence;
  rand RandClass ip_RandClass;

  function new(string name = "ipCfgSeq");
    super.new(name);
  endfunction

  virtual task body();
    `uvm_do(ip_RandClass)
  endtask: body
endclass
```

```
class ip_reconfig extends uvm_sequence;
  ipCfgSeq ipCfgSeq_h; //IP Config Wrapper Sequence --> Same Configurable Sequence
  `uvm_object_utils(ip_reconfig)

  virtual task body();
    super.body();
    // Polling on IP quiesce state
    wait(cfg.quiesce_state_attained);
    //During the reconfiguration flow, we disable the randomization of register regA.
    cfg.cfgSeq.regA_rand_en = 'h0;
    `uvm_do(ipCfgSeq_h) //Essentially this calls the RandClass, maintains all variables
    // Additional reconfiguration logic can be added here
    ..
  endtask: body

endclass: ip_reconfig
```

```
class user_func_seq extends base_sequence;
  `uvm_object_utils(user_func_seq)
    ip_reconfig ip_reconfig_h;
    ...

  virtual task body();
    super.body();
    `uvm_do(pm_flow_seq1)  //PM State Transistion
    `uvm_do(pm_flow_seq2)  //PM State Transistion
    `uvm_do(ip_reconfig_h) // Quiesce and reconfigure
    `uvm_do(pm_flow_seq3)  //Resume PM State Transistion
  endtask: body

endclass: user_func_seq
```

## Results

| Aspect | Traditional (No Unified Sequence) | Unified Configurable Sequence |
|---|---|---|
| Time to implement new configuration variant | ~1 hour manual sequence creation | ~5 minutes by deriving a new sequence class and overriding via UVM set_type |
| Effort to add new register | Manual sequence and object updates | Semi-automated: populate config object + base methods from register description |
| Sequence reuse across reset and runtime | Separate sequences required | Single unified sequence reused for both reset and runtime reconfiguration |
| Support for distinct reset vs. runtime randomization | Not supported | Supported: independent randomization for reset and runtime |
| Architectural flavor creation | Not supported | Supported: config object enables architectural variants |
| Configuration customization in tests | Manual per-test coding | Tests reuse unified config object with randomization/customization |

## Conclusion

- Unified hierarchical sequence (Base/Random/Custom) streamlines register programming and IP reconfiguration.
- Supports dynamic reconfiguration and asynchronous reset flows.
- Reduces development effort, eliminates duplicate sequences, and improves coverage.
- Broadly applicable across configurable IP architectures.

### Future Scope

- Automation of sequence generation from register specifications.
- AI-driven scenario selection & coverage optimization.
- Increased automation of config-object creation for scalability.

The authors thank Intel Corporation and Power Management IP teams for their contributions to the development and verification of this methodology.

Author's Contact : bhaskar.vedula@intel.com, stephen.p.haake@intel.com, chandrakanth.n.betageri@intel.com, ganesh.sharma@intel.com