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

Bhaskar Vedula
Intel Corporation
Portland, OR, USA
bhaskar.vedula@intel.com

Stephen P. Haake
Intel Corporation
Saint Paul, MN, USA
stephen.p.haake@intel.com

Chandrakanth N. Betageri
Intel Corporation
Santa Clara, CA, USA
chandrakanth.n.betageri@intel.com

Ganesh Sharma
Intel Corporation
Fort Collins, CO, USA
ganesh.sharma@intel.com

Abstract

Increasing configurability in IP architectures poses significant challenges for verification engineers. This paper introduces a modular and scalable testbench methodology utilizing a unified random configurable sequence pattern, hierarchical sequence abstraction, and factory override mechanisms built on native UVM principles. Applied to server CPU Power Management IP, this approach reduces development effort, enhances reuse, and improves coverage while supporting dynamic state transitions. The methodology is broadly applicable to complex designs requiring extensive register configuration and reconfiguration capabilities.

Index Terms

UVM; Register Programming, IP Verification; Testbench Infrastructure; Sequence Design Patterns, Power Management IP.

1 Introduction

Highly configurable IP architectures form the backbone of modern SOCs, enabling reuse and flexibility. The ability to modify IP behavior dynamically — at reset and runtime — has become crucial for achieving performance and power goals. However, this configurability introduces new verification challenges in managing numerous configuration registers, modes, and dynamic state transitions.

Traditional approaches, such as static configuration initialization combined with register randomization, focus on setting up the DUT for a single configuration. These techniques cannot dynamically reconfigure the IP during runtime, and they lack customization for specific scenarios.

This paper introduces a configurable sequence pattern combined with hierarchical sequence design techniques for programming the DUT registers. The proposed methodology enables the reuse of a single configuration sequence in initial reset, warm reset, and runtime windows. It also demonstrates how constrained derivatives of a highly randomized base sequence can be generated using native UVM factory override techniques without modifying the original sequence. This approach significantly reduces the test bench's development effort, enhances configurability, and improves the verification efficiency across complex IPs.

2 Related Work and Motivation

Traditional verification methodologies often rely on static configuration sequences and basic register randomization. While sufficient for initial IP bring-up, these approaches fall short in dynamic runtime reconfiguration scenarios. Standard UVM register sequences typically randomize the register model once at the beginning of the simulation (static configuration). This limits the ability to verify dynamic state transitions where the IP must be reconfigured on-the-fly without restarting the simulation.

Common industry practices involve using virtual sequences paired with configuration objects to manage testbench settings. However, these are often designed for static build-time or run-time configuration of the testbench topology, rather than for driving dynamic register programming sequences that can be reused across different phases. Consequently, verification teams often maintain separate sequences for initial boot and runtime reconfiguration, leading to code duplication and maintenance overhead.

Although UVM factory overrides are widely used to substitute sequences, they are rarely employed to create constrained "flavors" of a single unified configuration sequence that adapts to both reset and runtime contexts. Furthermore, while phase jumping has been explored in SoC bring-up papers for handling resets, integrating it with a unified hierarchical sequence pattern for seamless reconfiguration remains a gap in prior art.

Key challenges with current approaches include:

- **Static Nature:** Lack of support for dynamic reconfiguration during a single simulation run using standard register sequences.
- **Redundancy:** Requirement for duplicate sequences to handle initial setup versus runtime configuration.
- **Limited Flexibility:** Inability to easily customize or override randomization constraints dynamically for specific runtime phases without modifying the core sequence.

To address these limitations, a verification methodology is needed that unifies configuration handling, supports dynamic reconfiguration via phase jumping, and leverages hierarchical factory overrides to maximize reuse and flexibility.

3 Methodology Overview

3.1 UVM Phasing and Power Management IP flow

In complex IP architectures, there is an increasing need to dynamically modify configurations and functionality. In Power Management (PM) IP, we introduced a new configuration methodology that adapts IP configurations and functionality based on workload and architectural requirements. Our Power Management IP is responsible for controlling the power state transitions of various other IPs on the SoC, and therefore has unique configuration considerations. To limit verification permutations, some configuration registers can be changed at any time, while others can only be changed when power management flows are paused, and some require a hardware reset. Fig. 1 shows an overview of the dynamic configuration of the Power Management (PM) IP over time.

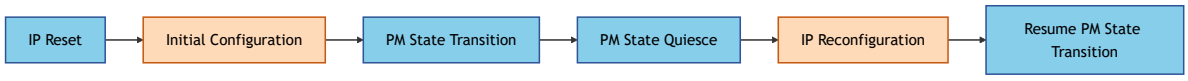

Fig. 1: Power Management IP Arch Flow

- **IP Reset:** Sets the hardware to its default state after the reset exit.
- **Initial Configuration:** Programs the registers to attain a legal architectural state.
- **PM State Transitions:** Initiates active and idle power management state transitions.
- **PM State Quiesce:** Inhibits future power state transitions and completes outstanding power state transitions.
- **IP Reconfiguration:** Modifies hardware registers to achieve a new architectural state.
- **Resume PM State Transitions:** IP resumes active and idle power management state transitions.

To implement the architectural flow illustrated in Fig. 1, we leverage the standard UVM phases with particular emphasis on the Run Phase, which is the only UVM phase where components execute concurrently and can be coordinated in orderly shutdown [1] [2]. Our PM IP verification framework extends a centralized verification architecture that supports custom UVM phases, enabling the integration of specific sequences to form the PM architectural flow.

Two distinct operational modes define the framework for integrating UVM phases with Power Management IP verification:

3.2 Dynamic Reconfiguration Mode

The dynamic mode of reconfiguration occurs in the Main Phase of the UVM Run Phase after the Power Management IP has exited reset. As shown in Fig. 2, an initial configuration is obtained by `configurable sequence` (label ①in Fig. 2). After initial configuration, power management algorithms and state transitions are initiated by UVM sequences (`PM State Transitions`) to test the desired scenario. Later, all desired power management stimuli have been applied, and there is a desire to reconfigure the Power Management IP. This requires the power management algorithms to pause and any in-flight power state transitions to complete via the `PM State Quiesce` UVM sequence. The Power Management IP is then reconfigured, reusing the same `configurable sequence` (label ②in Fig. 2). This dynamic reconfiguration process executes multiple iterations within a single UVM Main Phase, each representing different architectural operational modes, as shown in label C in Fig.2.

3.3 Dynamic Reset Mode

In this mode, the Power Management IP operates similarly to the Dynamic Reconfiguration Mode, but incorporates asynchronous reset handling capabilities. During active power management operations, the system may receive unexpected reset requests while managing other IPs. To address this scenario, asynchronous reset requests are injected to trigger controlled reset behavior without disrupting the overall verification flow.

This behavior is achieved through the advanced UVM Jump Phase technique shown in Fig. 2 label Ⓐ, which enables non-sequential phase transitions to handle these asynchronous conditions gracefully. The proposed methodology employs the same `configurable sequence` framework to reconfigure the IP across different architectural states, ensuring consistency between normal reconfiguration and reset-driven reconfiguration scenarios.

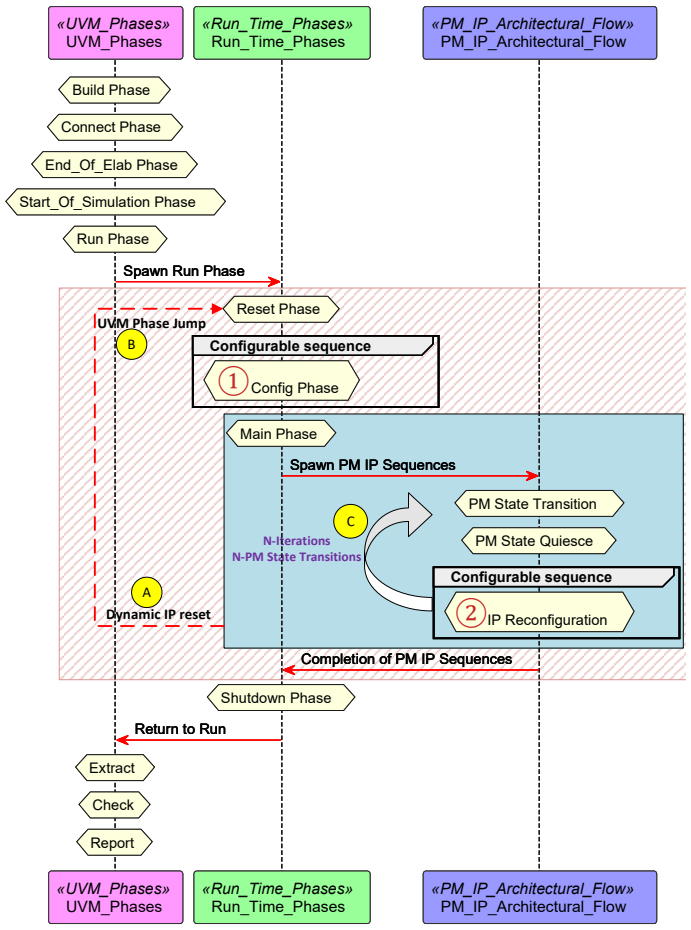

Fig. 2: Overlay Of UVM Phases With Power Management IP Arch flow

3.4 IP UVM Test

To achieve the behavior discussed above, Listing 1 shows the IP UVM test class that sets up the environment and connects the sequences to the respective UVM phases. The `configurable sequence` is invoked for both initial configuration and reconfiguration, demonstrating its versatility and reusability across different operational scenarios.

```
class ip_uvm_test extends uvm_test;

  function connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    env.set_test_phase_type( 'RESET_PHASE',   'reset_sequence');
    env.set_test_phase_type( 'CONFIG_PHASE',  'ipCfgSeq'); //Configurable Sequence
    env.set_test_phase_type( 'MAIN_PHASE',    'user_func_seq');
endfunction
  ......
endclass : ip_uvm_test
```

Listing 1: Generic UVM Test Class

3.5 Configurable Testbench Implementation

A unified IP `configurable sequence` is used within the testbench to manage both initial IP configuration setup and dynamic IP reconfiguration after reaching a quiescent state. Rather than creating separate sequences for initial bring-up and runtime operations, a single default random configurable sequence shown as ①in Fig. 3 is configurable through a sequence configuration object label ②in Fig. 3 to enable both use cases.

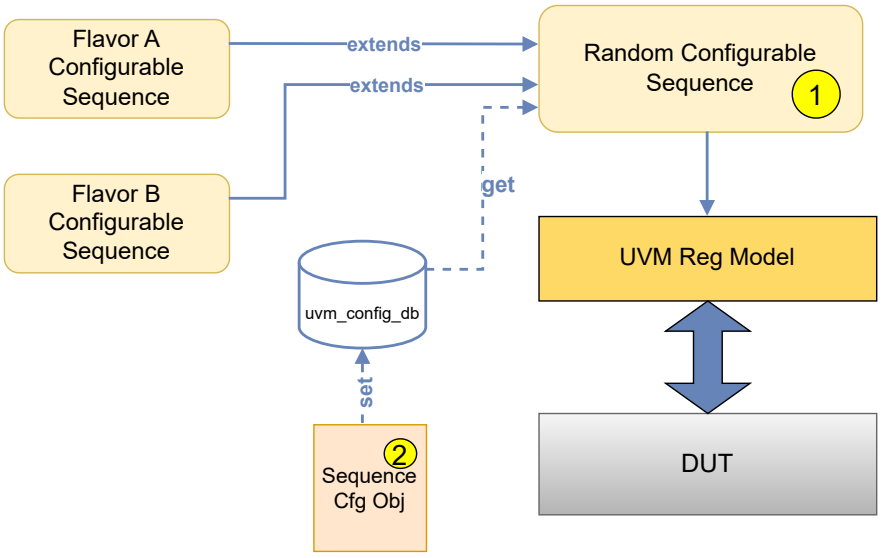

Fig. 3: Configurable Testbench Setup

The sequence configuration object defines random configuration variables that control randomization and generate register access based on the architecture configuration. This approach establishes layers of abstraction by developing the sequence in a hierarchical pattern, consisting of multiple layers as described below.

3.5.1 Configurable Sequence

The unified random IP `configurable sequence` introduced earlier serves as the foundation for accommodating diverse architectural configurations and managing complex data flow behaviors. To ensure scalability and modularity, the sequence employs constrained-random testing while maintaining ease of development and integration. A sequence object is used to randomize various elements of the reconfiguration flow, providing fine-grained control and targeted customization when required. By pairing this sequence with a configuration object, the methodology effectively enables a hierarchical sequence architecture, as illustrated in Fig. 4, offering flexible and efficient mechanisms for programming reconfiguration registers across different operational scenarios.

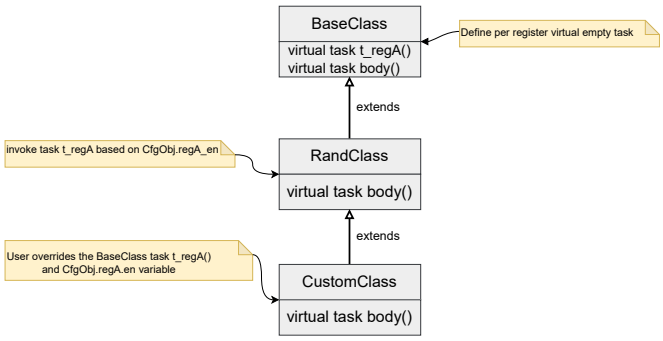

Fig. 4: Hierarchical Sequence Pattern

The proposed approach introduces a layered abstraction model that begins with a base sequence layer, which defines the fundamental IP DUT programming and functional constructs. The higher layers of the hierarchy build on this foundation to implement extended functionality and scenario-specific behaviors. Each subsequent layer is designed to handle increasingly complex and customized verification scenarios. This structured hierarchy efficiently manages architectural complexity, supports controlled randomization, and enhances configurability while providing verification engineers with fine-grained control over verification intent and stimulus generation.

3.5.2 Sequence Configuration Object

The sequence configuration object `seq_object` used with the unified configurable sequence serves as a central container for all control variables and default constraints required to drive the IP configuration. Each register or flow in the design is mapped

to a corresponding variable within this object, enabling precise control and traceability. To streamline sequence development, the object was automatically generated through a script that parsed the register flow output, minimizing manual effort and potential inconsistencies. Once created, the object is accessed within the sequence through the environment configuration, following standard UVM conventions. As illustrated in Listing 2 code snippet, the `seq_object` class definition allows verification engineers to easily define per-register configurations, adjust constraints, and customize behavior dynamically to meet specific feature coverage goals.

```
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
```

Listing 2: Sequence Configuration Object

3.5.3 Environment Configuration Object

Once the sequence configuration object is populated, we need to define the handle of the sequence configuration object within the environment configuration object so that all sequences and components can access the sequence configuration methods and variables. Listing 3 shows an example of the environment configuration object.

```
class env_cfg extends uvm_object;
  seq_object cfgSeq;
  ......
endclass : env_cfg
```

Listing 3: Environment Configuration Object

3.5.4 *Base Layer*

The base class is designed to access essential testbench resources, such as the register model and configuration object, by retrieving them from the testbench's config_db. Additionally, the base class defines virtual tasks for each design register or reconfiguration flow. These virtual tasks can be left empty or populated with default randomization, allowing users to override them as needed.

As illustrated in Listing 4, the base class demonstrates a typical `uvm_config_db::get()` operation to access the environment configuration within the sequence body. The example below highlights modular sequence construction encapsulating necessary virtual tasks, where the `t_regA()` task is left empty for user override, while the `t_regARand()` task implements default register randomization for baseline IP configuration.

```
class BaseClass extends uvm_sequence;
  `uvm_object_utils(BaseClass)
  virtual task body();
    if (!uvm_config_db #(env_config)::get(null, '*sqr', 'cfg', cfg)) begin
      `uvm_fatal(get_name(), `Did not 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
```

Listing 4: Base Class Sequence

3.5.5 *Random Layer*

The `RandClass` class is created in conjunction with the `seq_object` object handle as shown in Listing 5. The user defines the sequence of events in the sequence body, and the corresponding base class task is invoked based on the sequence configuration variable.

```
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
```

Listing 5: Random Layer Sequence

3.5.6 Custom Layer

As shown in Listing 6, users can apply override techniques to generate custom scenarios through the `CustomClass` class, which creates an abstraction layer for stimulus generation. Users can fully customize configuration values and override base class methods for specific verification scenarios.

The user requires full control over the `idle_power_management_policy` register field rather than complete randomization. To achieve this, the relevant sequence config object variables must be set: `cfgSeq.idle_power_management_policy_en` to 1 and `cfgSeq.idle_power_management_policy_rand_en` to 0. By configuring these variables appropriately, the corresponding base class task is overridden at this abstraction layer, and the customized task is subsequently invoked in the `RandClass` layer.

```
  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
```

Listing 6: Custom Layer Sequence

In Listing 7, the user creates a custom sequence for reset scenarios by programming the relevant configuration variables. In this example, the error severity can only be modified during reset conditions. Once this custom sequence is created, it provides the necessary framework for reset-specific verification scenarios.

```
class CustomResetClass extends RandClass;
  `uvm_object_utils(CustomResetClass)
  virtual task body();
    // enable programming of registers that shall not be reconfigured after reset
    cfg.cfgSeq.error_severity_control_en = 1;
    ...

  endtask: body
  task t_error_severity_control()
    cfg.regblock.error_severity_control.x_is_fatal.set(1);
    cfg.regblock.error_severity_control.y_is_fatal.set(1);// non-default value
    cfg.regblock.error_severity_control.update();
    `uvm_info(get_name(), `error_severity_control runtime reset programming`,
       UVM_LOW);
  endtask
endclass: CustomResetClass
```

Listing 7: Custom Reset Layer Sequence

3.5.7 Test Setup

The test setup demonstrates how the hierarchical sequence pattern integrates with the UVM test infrastructure. The `ipCfgSeq` wrapper sequence shown in the Listing. 8 provides a clean interface between the test environment and the configurable sequence hierarchy, while the test class establishes the proper phase-to-sequence mappings for coordinated execution.

For custom scenarios, the `ipCfgSeq` sequence instantiates `RandClass` by default, but users can override it with `CustomClass` via UVM type override to enable specialized behavior.

```
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
```

Listing 8: IP Config Wrapper Sequence

For example, to override `RandClass` with `CustomClass` from the command line, use:

```
+uvm_set_type_override=RandClass,CustomClass
```

This ensures that all instances of `RandClass` are replaced by `CustomClass` during simulation, allowing custom configuration and behavior without modifying the testbench code.

3.6 IP Reconfiguration Flow

In the IP reconfiguration sequence shown in Fig. 1, once the DUT enters a quiescent state, the targeted registers are reprogrammed to transition the IP into a new architectural state using either `RandClass` or `CustomClass`, thereby altering its functionality. To achieve this, a new sequence Listing 9 is written that references the IP Config Wrapper Sequence (see Listing 8). Since the execution occurs within a single Main Phase, the sequence can wait for the IP to reach the quiescent state, reconfigure the necessary registers, and then resume execution. Example Listing 10, shows the sequence that runs in main phase.

```
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
```

Listing 9: IP Reconfiguration Sequence

```
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
```

Listing 10: PM IP User Sequence

This approach enables the generation of different architectural scenario sequences and supports dynamic type overrides during the reconfiguration flow, making it easy to create various sequence flavors as needed.

By using the same configurable sequence for both initial configuration and reconfiguration, the methodology ensures consistency, simplifies maintenance, and preserves architectural integrity.

3.7 Dynamic Reset condition

The Dynamic Reset condition represents a critical verification scenario where asynchronous reset events occur during active Power Management operations. Unlike the controlled reconfiguration in Dynamic Reconfiguration Mode, reset conditions can be unexpectedly triggered at any time, requiring robust handling mechanisms to maintain verification integrity.

In real-world SOC environments, reset events may originate from various sources, including system-level reset controllers, thermal management units, or error recovery mechanisms. These events must be properly handled regardless of the current PM IP state, whether it is actively managing power transitions, processing configuration changes, or coordinating with dependent IP blocks.

The unified configurable sequence methodology provides effective support for handling reset conditions through several key mechanisms:

3.7.1 Asynchronous Reset Injection

Reset stimuli are injected asynchronously during active verification scenarios using dedicated reset agents. These agents monitor the verification environment and can trigger reset events based on predefined conditions, random intervals, or specific test scenarios. The reset injection mechanism operates independently of the main verification flow, ensuring realistic timing relationships between normal operations and reset events.

3.7.2 UVM Jump Phase Implementation

When a reset condition is detected, the verification environment employs UVM Jump Phase techniques to perform non-sequential phase transitions. This allows the testbench to immediately transition from any Main Phase sequence (e.g, PM State Transitions, Quiesce) directly to the Reset Phase without disrupting the overall verification architecture.

Following reset assertion and deassertion, the IP must be reconfigured to resume normal operation. The methodology leverages the same hierarchical configurable sequence pattern used in normal reconfiguration scenarios. This ensures consistency in register programming approaches and maintains the benefits of the unified sequence architecture even in reset recovery scenarios.

Custom reset initialization sequences, such as the one in Listing 7, can be reused in any reset phase iteration, or different versions of the sequence can be created for each reset phase iteration. This approach enables comprehensive testing of reset robustness across all supported architectural configurations.

4 Results & Discussion

The unified random configurable sequence was deployed in a server CPU Power Management IP, enabling both initial setup and runtime reconfiguration using a single hierarchical sequence and UVM configuration object. This approach allows for rapid creation of new sequence variants via inheritance and UVM overrides, reducing code changes and improving reuse. The methodology is applicable to other complex designs that require flexible register configuration. Table I summarizes the key benefits.

Table I: Technical Comparison: Unified Configurable Sequence vs. Traditional Approach

Aspect	**Traditional (No Unified Sequence)**	**Unified Configurable Sequence**
Time to implement new configuration variant	$\sim$ 2x: manual sequence creation	$\sim$1x: derive new sequence class and override via UVM `set_type`
Effort to add new register	Manual sequence and object updates	Semi-automated: populate config object and base methods from register description
Sequence reuse across reset and runtime	Separate sequences required	Single 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	All tests leverage unified config object with randomization/customization

5 Applicability

The unified configurable sequence methodology is broadly applicable to any IP that requires frequent configuration and reconfiguration. Its hierarchical pattern enables:

- **Easy Reuse:** Extendable to new IP revisions by updating configuration objects or constraints.
- **Rapid Bring-up:** Quickly adapts to new IPs via automated configuration object generation.
- **Improved Coverage:** Supports both random and targeted overrides for thorough scenario exploration.

6 Conclusion

This paper presents a comprehensive UVM-based verification methodology that successfully addresses the growing complexity of highly configurable IP architectures through a unified configurable sequence pattern. The approach enables dynamic reconfiguration capabilities while maintaining code reusability and reducing development effort.

The key contributions of this work include:

Unified Sequence Architecture: A hierarchical sequence pattern (Base, Random, Custom layers) enables scalable, modular, and reusable register programming for diverse configurations.

Dynamic Verification: Supports both runtime reconfiguration and asynchronous reset scenarios via coordinated UVM phasing and Jump Phase techniques.

Productivity Gains: Implementing new configuration variants is 2x faster, while achieving 100% sequence reuse between the reset and runtime phases.

Broad Applicability: Applicable to any configurable IP that requires flexible register programming and reconfiguration.

This methodology simplifies the verification of configurable IP's by enabling dynamic reconfiguration, efficient stimulus generation, and comprehensive coverage using a hierarchical unified UVM sequence pattern. It reduces development effort and code duplication while supporting complex state transitions and runtime scenarios. Future work includes automating sequence generation from register specifications and applying AI for scenario selection and coverage optimization, further enhancing scalability and applicability.

Acknowledgment

The authors thank the members of the Intel Corporation verification team who contributed to the development and validation of this methodology. Special appreciation goes to the Power Management IP development team for providing the production environment for testing and validation, and to the engineering managers who supported the implementation of this approach in complex IP designs. We also acknowledge the valuable feedback from the DVCon Technical Program Committee (TPC) that helped improve the quality and clarity of this work. The authors thank Intel for its support in enabling this research and facilitating its publication for the benefit of the broader verification community.

References

[1] Mentor Graphics, Online UVM Cookbook, "UVM RAL Section," https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/integrating-a-uvm-register-model-in-a-testbench-overview/

[2] Accellera Systems Initiative, "Universal Verification Methodology (UVM) 1.2," https://accellera.org/downloads/standards/uvm, 2017.