# Random Directed Low Power Coverage Methodology: A Smart Approach to Power Aware Verification Closure

Awashesh Kumar, Mentor Graphics (awashesh_kumar@mentor.com)

Madhur Bhargava, Mentor Graphics (madhur_bhargava@mentor.com)

*Abstract* **-With the advancement in the technology, the size and complexity of SoC is ever increasing. At the same time the power constraints on those SoC are getting more stringent. The low power designs are getting more and more complex with large number of power domain, supplies and power states. This makes the job of low power SoC verification overwhelmingly complex, tedious and sophisticated. It is the need of the hour to develop a reliable verification methodology which is easy to build, easy to understand and easy to modify. UPF 3.0 – The IEEE standard to specify the low power intent, has introduced a concept of low-power information model which can be useful to address the complex verification challenges. This paper introduces the key concepts of UPF 3.0 information model which are useful from low-power verification point of view. The paper proposes a low-power coverage methodology based on the UPF 3.0 information model HDL package. The paper also includes relevant case studies and examples using the proposed methodology to solve low-power verification problems. It also discusses the benefits of this approach and its advantages over conventional low-power verification approaches. There are still some areas like analog and mixed signal low power verification where the proposed coverage methodology cannot be used.**

**Keywords - Power Management, Power Aware Verification, Low-Power Coverage Methodology**

## I. INTRODUCTION

With the advancement in the technology, low-power designs and its verification is becoming more complex. Today's chips have multiple power domains each having multiple operating power modes and dynamically changing voltage levels. The power architecture is controlled with help of a power management unit which issues proper control sequences to different elements of the power architecture. It is important to ensure that all and proper test vectors are generated by this unit to verify all power elements and it is even more important to verify the complex interactions between these elements at a higher abstraction level. A plan to closure approach for low-power verification requires coverage of power objects (all possible states and transitions). However power-aware coverage closure is hard and complex in nature.

It is important to note that there is no standardized methodology for low-power coverage, as UPF 2.1 doesn't provide anything to address this. With the help of current UPF standards, verification engineers have taken the ad-hoc approach to achieve coverage of some constructs and their interactions with each other e.g. power states etc. One such technique is cross-coverage of power states. However it is prone to state explosions and it involves low level intricate efforts to write the covergroups and coverbins for the cross-coverage. Verification engineer needs to be well aware of the UPF to figure out the supplies and signals to track. Moreover this whole process is tedious and dependent on verification tools and EDA vendors. The whole process is error prone and highly time consuming. So a faster and direct approach to address low power coverage is a need of the hour.

To keep pace with that, IEEE 1801 standard is expanding its gamut of constructs and commands to include more scenarios of low-power verification and implementation. In this paper, we will discuss and propose a methodology by which effective coverage of low-power verification can be achieved using UPF 3.0. With the help of relevant examples and case studies we will also demonstrate that with the help of this methodology, the coverage closure can be achieved in much efficient way thereby significantly saving the verification effort and time.

### *A. Power Intent Specification and Basic Concepts of UPF*

IEEE Std 1801™-2015 Unified Power Format (UPF) allows designers to specify the power intent of the design. It is based on Tcl and provides concepts and commands which are necessary to describe the power management requirements for IPs or complete SoCs. A power intent specification in UPF is used throughout the design flow; however it may be refined at various steps in the design cycle. Some of the important concepts and terminology used in power intent specification are the following:

- Power domain: A collection of HDL module instances and/or library cells that are treated as a group for power management purposes. The instances of a power domain typically, but do not always, share a primary supply set and typically are all in the same power state at a given time. This group of instances is referred to as the extent of a power domain.

- Power state: The state of a supply net, supply port, supply set, or power domain. It is an abstract representation of the voltage and current characteristics of a power supply, and also an abstract representation of the operating mode of the elements of a power domain or of a module instance (e.g., on, off, sleep).

- Isolation Cell: An instance that passes logic values during normal mode operation and clamps its output to some specified logic value when a control signal is asserted. It is required when the driving logic supply is switched off while the receiving logic supply is still on.

- Level Shifter: An instance that translates signal values from an input voltage swing to a different output voltage swing.

- Retention: Enhanced functionality associated with selected sequential elements or a memory such that memory values can be preserved during the power-down state of the primary supplies.

## II. MOTIVATION FOR METHODOLOGY

In order to achieve comprehensive low-power verification, a verification engineer is interested in a number of questions, such as:

- If all the test vectors are generated by power management unit ensuring that all control sequences are covered
- Are all complex interactions between power domains are covered?
- Are all the desired power states are reached or not?
- Are all desired power state transitions reached or not?
- Is there any illegal power state reached?
- Is there any illegal power state transition occurred?

These kinds of questions are easily addressed by coverage-driven verification. But, it becomes a challenge to capture the coverage information of UPF objects and power states due to following reasons:

- Power states are written in an abstract manner in UPF, and
- There is no pre-defined coverage metric to capture power states and their transitions.
- The Unified Coverage Interoperability Standard (UCIS) defines various standard metrics related to coverage items. However, it does not provide any metric to capture power intent (power states, etc.).

Tool-generated coverage metric are used widely for low-power verification. However they may not be useful or exhaustive in all the designs, as highlighted by the reasons below:

- A design can have a very specific requirement which is not being provided by the tool-generated coverage.
- The low-power technology is still evolving and hence a new set of protocol appears every now and then, which may require a different set of coverage that is not yet provided by the tool vendor.

These requirements can be easily met through SystemVerilog functional coverage features. The covergroup construct of SystemVerilog samples the activities of a signal/property at desired sampling points and one can distribute these sample points under different coverpoints and bins. These coverpoints and bins can be effectively

used to collect coverage numbers of power states and their transitions. Covergroups also offer a generous syntax to handle a wide range of complex sampling scenarios especially the wildcard feature for handling state transitions when multiple states are true at the same time.

Due to the above reasons the user may want to write custom coverage items using the System Verilog coverage features. These items can be grouped in a checker module, and this checker module can be instantiated into the design using UPF command "`bind_checker`". However the method of instantiation of such a checker module is not trivial because:

- These low-power assertions/coverage items require access to power objects. However, at the early stages of verification these power objects are only present in the UPF file and do not exist in the design. It is therefore not straightforward to pass these UPF objects to a checker instance.
- No clear way to describe coverpoints and bins to capture coverage of power states and their transitions using the handles obtained
- Some of the property-checking requires access to design/power signals spanning across multiple domains. Such a task is highly error prone and time consuming.
- As the scope and the inputs of these checkers instances are defined in UPF, any change in the UPF or design might break these checkers and they need to be re-written.

We propose a modeling style using covergroups and UPF's information model query commands command to address these challenges.

## III. UPF 3.0 INFORMATION MODEL

UPF 3.0 has introduced the low power information model to represent the low power objects created in UPF e.g. power domain, power state, supply sets etc. It also provides the detailed list of various properties which those objects have. These properties can be a simple information e.g. the name of the object, the file/line information of the object or a relatively complex information e.g. the power states of a power domain. For some objects there is also dynamic properties associated with them. For example the current_state of a power domain, or the current voltage value of a supply net. UPF 3.0 information model provides an API interface to access the objects and its properties. There are two kinds of API interface provided in UPF 3.0

- Tcl interface; to use the information model APIs in a Tcl script or UPF file
- HDL interface; to access and manipulate information model objects/properties in a testbench or simulation model

With respect to the paper, HDL interface and the dynamic properties of the objects are more relevant. There are two key concepts of information model which can be used to leverage it to develop a low power coverage methodology which can be random in construction and directed in approach to target specific low power scenarios.

### *A. Native HDL representation*

UPF 3.0 defines the native HDL representation for the objects which have the dynamic properties. The native HDL representation is the struct/record type in HDL that contains two fields.

- A value field corresponding to dynamic property of the object.
- A handle or reference to the UPF object. To allow access of other properties of the object.

Following HDL types are supported with a native HDL representation:

Table 1.

| **Type Name** | **SV Representation** |
|---|---|
| upfPdSsObjT | struct {<br>upfHandleT handle;<br>**upfPowerStateObjT current_state;**<br>} upfPdSsObjT |

| | |
|---|---|
| upfPowerStateObjT | struct {<br>upfHandleT handle;<br>**upfBooleanT is_active;**<br>} upfPowerStateObjT |
| upfBooleanObjT | struct {<br>upfHandleT handle;<br>**upfBooleanT current_value;**<br>} upfBooleanObjT |
| upfBooleanObjT | struct {<br>upfHandleT handle;<br>**upfSupplyTypeT current_value;**<br>} upfSupplyObjT |

In the Table 1 above the field representing the dynamic property of the object has been highlighted in bold. E.g. for a power domain or supply set the associated dynamic property is the current power state of the power domain which is represented by the current_state field of the struct in SV native representation of upfPdSsObjT type. The other field is a handle to the low power object which has all the static information about the object e.g. object name, its creation scope, file/line information etc.

Following table 2 summarizes the UPF 3.0 information model objects with native HDL information. The HDL types defined in table 1 are used to represent the dynamic properties of these objects.

Table 2

| **Low Power Object Type** | **Dynamic Property** | **Low Power Idea Represented** | **Native HDL Type** |
|---|---|---|---|
| upfPowerDomainT | current_state | Current power state | upfPdSsObjT |
| upfSupplySetT | current_state | Current power state | upfPdSsObjT |
| upfCompositeDomainT | current_state | Current power state | upfPdSsObjT |
| upfPstStateT | is_active | Is the PST currently active | upfPowerStateObjT |
| upfPowerStateT | is_active | Is the power state currently active | upfPowerStateObjT |
| upfAckPortT | current_value | Logic value at the port | upfBooleanObjT |
| upfExpressionT | current_value | Value of the expression | upfBooleanObjT |
| upfLogicNetT | current_value | Logic value of the net | upfBooleanObjT |
| upfLogicPortT | current_value | Logic value of the port | upfBooleanObjT |
| upfSupplyNetT | current_value | Value of the supply net | upfSupplyObjT |
| upfSupplyPortT | current_value | Value of the supply port | upfSupplyObjT |

***B.*** *HDL package functions*

UPF 3.0 provides a number of HDL package functions which are used to access the low power objects and their properties. These are broadly classified in the following five different class of functions.

1. HDL access functions – These are the basic functions to access the low power objects and properties. E.g. following access function can be used to get the handle of an object.

   *upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd")* - returns the handle of the power domain 'pd'.
   One of the key HDL access function is the "upf_query_object_properties".
   *upfHandleT upf_query_object_properties(upfHandleT object_handle,upfPropertyIdE attr);*
   This function returns the handle to a property corresponding to an enumerated value passed as property.
   E.g. *upfHandleT scope = upf_query_object_properties(pd, UPF_CREATION_SCOPE)* - returns the creation scope of power domain with handle 'pd'.

2. Immediate read access HDL functions: All the objects in UPF 3.0 information model allow read access to its properties. In case of dynamic properties these functions return the current dynamic value/state or that property when this function is called. E.g.

   *upfHandleT ps = upf_get_handle_by_name("/top/dut_i/pd.power_state_on")*
   *upfHandleT ps_active_hndl = upf_query_object_properties(ps, UPF_IS_ACTIVE )*
   *integer ps_on_value = upf_get_value_real(ps_active_hndl)*

3. Immediate write access HDL functions: Some objects of the information model allow the immediate write access only if they don't have an existing driver. This allows the manipulated of low power objects from testbench or simulation model. E.g. supply_on("supply_net_name", value). Following objects which allow immediate write access
   a. upfPowerStateT
   b. upfLogicNetT
   c. upfLogicPortT
   d. upfSupplyNetT
   e. upfSupplyPortT

In the context of this paper write access functions are not much relevant, although they are a powerful tool for users to manipulate low power objects during simulation from testbench.

4. Continuous access HDL functions – These functions enable continuous monitoring of dynamic values of an object in the information model. It enables user to trigger an always block or process statement using dynamic values of the low power objects.

   E.g. *upfSupplyObjT vdd_monitor;*
   *upf_create_object_mirror("/top/dut_i/vdd", "vdd_monitor");*

5. Utility functions: These functions are general utility function to assist users.
   E.g. *upfClassIdE upf_query_object_type(upfHandleT handle)* – returns the type of a handle, using this user can find out if the object is a power domain, supply set or some other low power object

## IV. Proposed Low Power Coverage Methodology

The UPF 3.0 HDL functions can be combined with System Verilog coverage constructs to devise an efficient and directed low power coverage methodology. The aim of such low power coverage methodology is to enable the users to write fast and reliable low power coverage infrastructure. It allows users to do the random (testing scenarios can be developed by some generic script) and directed (it can cover a very specific scenario) low power verification. In this paper we will propose a methodology to use the UPF 3.0 package functions to achieve fast and effective low-power coverage using relevant examples and case studies

The proposed coverage methodology involves getting the handle of low-power object in HDL using the information model and querying the properties of that object. These properties are then further passed to the coverage module as port mapping. The coverage module has the covergroups/coverbins to represent the coverage data of low-power strategies of that domain or dynamic information like the current power state of the domain can be queried in testbench.

Steps involved in the proposed methodology:

1. Low power object handle: First step is to get the handle of the low power object. This can be achieved by using the HDL package function *"upf_get_handle_by_name".* A property which is a list can be iterated through the HDL access function *"upf_iter_get_next".* Following SV code illustrates the usage of these basic function to access low power objects.

   ```
   upfHandleT pd = upf_get_handle_by_name("/tb/dut/pd")
   ```

```
upfHandleT pd_state_list = upf_query_object_properties(pd,
                                                    UPF_PD_STATES)
upfHandleT pd_state = upf_iter_get_next(pd_state_list);
```

2. Dynamic property(value) of low power object: Get the dynamic value of the property of interest e.g. user might be interested in knowing the current power state of a power domain and he intends to check the state of this power domain w.r.t. the current power state of some other power domain. Also user may use UPF 3.0 continuous access HDL package function to continually monitor the power state of the power domain and may use assertions for the cases of interest/anomaly.

```
upfPdSsObjT pd_hdl;
upf_create_object_mirror("/tb/dut/pd", "pd");
upfPowerStateObjT pwr_state = pd.current_state;
upfHandleT pd_name = upf_query_object_properties(pd.handle, UPF_NAME);
upfHandleT state_name = upf_query_object_properties(pwr_state.handle,
                                                    UPF_NAME);
always@(pd)
         $display( "Power domain %s, is in Power State: %s,
         upf_get_value_str(pd_name), upf_get_value_str(state_name));
```

3. Coverage details: Once we have the handle of a low-power object and its dynamic value it is further passed to a coverage module. The coverage module can be modeled in system Verilog and compiled together with the design. This module is instantiated in the testbench. The low-power object extracted from information model and its dynamic properties are passed in the interface of this coverage instance. We can instantiate as many coverage instances as required.
    - Coverage properties (covergroups/coverpoints) are defined inside this coverage module to calculate the coverage metrics.

```
module covIsoModule (int dynamicValue, string objName)
reg cov_clk = 0;
covergroup LOW_POWER_STATE_COVERAGE(posedge @cov_clk)
      ACTIVE_LEVEL: coverpoint isovalue { bins ACTIVE = 1; }
      ACTIVE_LOW: coverpoint isovalue { bins ACTIVE = 0; }
endgroup
…
endmodule
```

## V. CASE STUDIES AND EXAMPLES

### *A. Coverage of Isolation strategies*

In a large SoC design comprising of several power domains, each with a number of strategies defined on them, can often result in a scenario where multiple strategies affect a domain crossing. This can result in the placement of multiple isolation cells in the design. It becomes critical to ensure that all the possible states of isolation strategies are covered. This boils down to checking that all possible values of isolation control are covered.

Following is an example how a user can get the isolation control of a signal without going deep into UPF to find the isolation control. This is further passed onto a coverage module to calculate coverage. We first describe the coverage module which is used to calculate coverage of isolation control. This coverage module is written separately and compiled together with the design.

**UPF**

```
set_isolation iso –domain pd –isolation_signal isoCtrl \
      –applies_to outputs –clamp_value 0 …
```

**Coverage module**

```
module covIsoModule (int isovalue, string isoName)
      covergroup ISO_SIG_STATE_COVERAGE(@isovalue)
            ACTIVE_LEVEL: coverpoint isovalue { bins ACTIVE = 1; }
            ACTIVE_LOW: coverpoint isovalue { bins ACTIVE = 0; }
       endgroup
…
endmodule
```

The above covergroup contains two coverpoint; ACTIVE_LEVEL is reached when the isolation control is "1" and ACTIVE_LOW is reached when isolation control is "0".

Following is the hdl code which has to be modeled in the testbench. The first step is to get the handle of isolation strategy and thus its isolation control. Next is to monitor the value of isolation control. We can instantiate multiple coverage instances for each isolation strategy and pass the value of isolation control to these coverage instances.

**HDL Code (modeled in testbench)**

```
upfHandleT iso = upf_get_handle_by_name("/top/chip_top/u_mod/pd.iso");
upfHandleT iso_ctrl = upf_query_object_properities(iso,

                                                    UPF_ISOLATION_CONTROLS);
upfHandleT ctrl_value_hndl =
                  upf_get_object_properties(iso_ctrl,UPF_CURRENT_VALUE);
integer ctrl_value = upf_get_value_int(ctrl_value_hndl);

// passing the properties to coverage module
covIsoModule covPdIso(ctrl_value, "/top/chip_top/u_mod/pd.iso");
```

*B. Coverage of Power States*

In typical low-power designs with multiple power domains, each domain can transition into multiple power states. As a result the total number of power states in a design can be very high. It becomes critical to ensure that all the power states of every power domain have been active at least once during the entire simulation. Also it is important to ensure that all the valid power state transitions have been covered. As the power states are defined in an abstract manner in UPF file, it is difficult to do coverage of power states. Moreover there is no standard way defined for capturing the coverage of power states.

Let's consider the following example where we have a camera domain (PD_CAMERA) with three power states "ON, SLEEP, OFF" and a video domain (PD_VIDEO) with three power states "ON, SLEEP, OFF". These power state definitions are defined in UPF using the "add_power_state" command.

```
add_power_state PD_CAMERA \
      -state ON {-logic_expr {primary == ON} } }
      …
```

First step is to extract the handles of power states of PD_CAMERA and get the state variable for these states which gets active when the particular power state is high. This is achieved with following Information model API calls defined in testbench. These state variable are then passed on to the coverage instance.

```
// Native HDL objects for power states
upfPowerStateObjT ps_state_ON;
upfPowerStateObjT ps_state_OFF;
upfPowerStateObjT ps_state_SLEEP;
```

```
// Handle to hold active state
upfHandleT state_ON_active;
upfHandleT state_OFF_active;
upfHandleT state_SLEEP_active;

//integer values of active states, 1 indicates active
integer state_ON;
integer state_OFF;
integer state_SLEEP;

// continuous access of power states
upf_create_object_mirror ("/tb/chip_top/PD_CAMERA.ON", "ps_state_ON")
upf_create_object_mirror ("/tb/chip_top/PD_CAMERA.OFF", "ps_state_OFF")
upf_create_object_mirror ("/tb/chip_top/PD_CAMERA.SLEEP", "ps_state_SLEEP")



always @(ps_state_ON, ps_state_OFF, ps_state_SLEEP) begin
      state_ON_active = upf_get_object_properties (ps_state_ON.handle,
                                                   UPF_IS_ACTIVE)
      state_OFF_active = upf_get_object_properties (ps_state_OFF.handle,
                                                   UPF_IS_ACTIVE)
      state_SLEEP_active = upf_get_object_properties (ps_state_SLEEP.handle,
                                                   UPF_IS_ACTIVE)

      state_ON = upf_get_value_int(state_ON_active);
      state_OFF = upf_get_value_int(state_OFF_active);
      state_SLEEP = upf_get_value_int(state_SLEEP_active);
end
```

These state variables state_ON, state_OFF, state_SLEEP are used to collect state coverage information. In order to collect transition coverage information we define an array with state variables as elements; we call this an array transition variable. Any change in any of the state variables will trigger a clock which will act as the sampling event for state and transition coverage. State determining logic in our sample example looks like:

**Coverage module**

```
module covPowerStateModule (int state_ON, state_OFF, state_Sleep, string
StateName)
wire [2:0] curr_state;
reg cov_clk = 0;
assign curr_state = {state_OFF, state_ON, ……};

always @(state_OFF, state_ON, ……)
      cov_clk = 1'b1;

always @(cov_clk)
      cov_clk = 1'b0;
endmodule
```

We define covergroups for state and transition coverage. The covergroup modeling of the state coverage has a number of coverpoints that sample the state variables. The covergroup modeling of the transition coverage has a coverpoint with bins that sample various transitions of the transition variable. State and transition covergroups for our simple example appear as follows:

```
// covergroup modeling of state coverage
covergroup PD_CAMERA_STATE_COVERAGE @(posedge cov_clk);
OFF: coverpoint state_OFF
{
 bins ACTIVE = (0=>1);
}
ON: coverpoint state_ON
{
 bins ACTIVE = (0=>1);
}
SLEEP: coverpoint state_SLEEP
{
 bins ACTIVE = (0=>1);
}
……
endgroup

PD_CAMERA_STATE_COVERAGE PS_primary = new;

// covergroup modeling of state transition coverage
covergroup primary_TRANSITION_COVERAGE @(posedge cov_clk);
type_option.strobe = 1;
PA_CAMERA_TRANSITION_COVERAGE:coverpoint curr_state
{
      wildcard bins OFF_to_ON = (3'b??1=> 3'b?1?);
      wildcard bins OFF_to_SLEEP = (3'b??1=> 3'b1??);
      wildcard bins ON_to_OFF = (3'b?1?=> 3'b??1);
      ……
}
endgroup
```

In the similar manner, we can achieve the coverage of PD_VIDEO.

*C. Assertion with UPF 3.0 information model*

In the above example defined in section B, there could be a requirement that the combination of state ON for PD_CAMERA and state ON for PD_VIDEO cannot be true at the same time. This can be checked easily using the proposed coverage methodology approach.

We can define an assertion module instead of a coverage module and create an instance of this module in testbench.

**Assertion Module**

```
module assertionPowerState (int state_ON_CAMERA, int state_ON_VIDEO)
reg cov_clk = 0;
always @(state_ON_CAMERA, state_ON_VIDEO)
      cov_clk = 1'b1;

always @(cov_clk)
      cov_clk = 1'b0;

always@(posedge cov_clk)
      assert (state_ON_CAMERA != state_ON_VIDEO) else $error("Camera and
Video both on at same time");
endmodule
```

## VI. Benefits of Proposed Power Coverage Methodology

Verifications engineers can use the proposed verification approach to achieve an early low-power coverage closure. It is possible to do a directed scenario testing using this methodology. For example user can write a custom assertion or property to check if two domains should be in a mutually exclusive state by using UPF 3.0 information model immediate read access function. Since the methodology relies on UPF constructs, so it is consistent and usable across different vendor tools. Even without detailed knowledge of complex UPF constructs and semantics, verification engineers can write various directed verification scenarios to achieve low-power coverage closure. The proposed approach can be easily scaled to bigger and more complex low power design scenarios. The proposed approach can be compared with approach mentioned in [4] where the verification engineer needs to be aware of the low level details like supplies and logic values of constituents of supply and logic expressions of power states. Whereas in the examples presented above such low level details are not required. In the same way cross coverage of power states as mentioned in [3] can be easily mapped to current approach.

## VII. Unaddressed Low Power challenges & Future Work

The current approach mentioned in this paper is versatile and robust however it lacks few fundamental things like ability to access analog/mixed-signal parameters from onboard regulators & voltage sources – as these are not part of the HDL or UPF. However it is an important part of the design and cannot be ignored.

## VIII. Conclusion

The complexities in low power verification have greatly increased. Any ad-hoc approach is not likely to succeed and will result in incomplete verification. In this paper, we proposed a methodology for low-power coverage of the design using the UPF 3.0 HDL package functions. This methodology will help achieve verification closure in significantly lesser time. We presented case studies and examples to elaborate the usage of the UPF 3.0 HDL package functions with System Verilog coverage constructs like covergroups and coverpoints to achieve the coverage of some key design failure scenarios of a low power design. This paper elaborated on the benefits of using the proposed approach over other conventional approaches. The paper also touched briefly on the future work and challenges in using the proposed method.

## References

[1] IEEE Std 1801™-2015 for Design and Verification of Low Power Integrated Circuits. IEEE Computer Society, 05 Dec 2015.
[2] “Amit Srivastava, Awashesh Kumar”, PA-APIs: Looking beyond power intent specification formats, DVCon USA 2015
[3] “Veeresh Vikram Singh, Awashesh Kumar”, Cross Coverage of Power States, DVCon USA 2016
[4] “Pankaj Kumar Dwivedi, Amit Srivastava, Veeresh Vikram Singh”, Let’s DisCOVER Power States, DVCon USA 2015