Low Power Apps:
Shaping the Future of Low Power Verification

Awashesh Kumar (Awashesh_kumar@mentor.com), Mentor, A Siemens Business
Madhur Bhargava (Madhur_bhargava@mentor.com), Mentor, A Siemens Business
Vinay Singh (vinay_singh@mentor.com), Mentor, A Siemens Business
Pankaj Gairola (Pankaj_gairola@mentor.com), Mentor, A Siemens Business

Abstract – **The current generation of SoCs are incredibly complex. The low-power architecture used in today's chips are even more sophisticated and the future trend is only going to go in the same direction. Efficient low-power architectures have become a necessity. Low-power requirements of SoCs have become as critical as functionality or timing. The complexity of low-power architecture places an enormous burden on the verification engineers. It is now very crucial to catch any power bugs early in the design cycle. Unified Power Format (UPF), the IEEE standard for low-power specification is constantly evolving to address the low-power requirements of the designs. Going in the same direction IEEE 1801-2015 (aka UPF 3.0) introduced the concept of a low-power information model which can greatly simplify the life of verification engineers by providing HDL and Tcl APIs to access and manipulate the low-power information. In order to debug the low-power issues and verify the design efficiently and timely, it has become of utmost importance to come up with a new approach. The paper describes how verification and design engineers can innovatively make use of these UPF 3.0 information model-based HDL and Tcl APIs to write useful low-power apps. In this paper we are also going to propose some of low-power apps which can be used to solve complex verification issues. We have also presented case studies and examples to demonstrate such usage. The paper also discusses the benefits of using the aforementioned approach.**

I. INTRODUCTION

The effective verification of low-power designs has been a challenge for many years now. The IEEE Std 1801™-2015 Unified Power Format (UPF) standard for modeling low-power objects and concepts is continuously evolving to address the low-power challenges of today's complex designs. One of the main challenges for low-power verification engineers has been the fact that there is a disconnect between the traditional RTL and low-power objects. Users cannot access and manipulate the low-power objects in the same way as they do for RTL. Low-power concepts are abstract and complexities arise because of the number of sources like UPF, HDL and Liberty all provide power intent in a low-power design. It has also been seen that the majority of verification time is spent debugging complex low-power issues. There are not too many ways in which users can do self-checking of their designs. As the low-power architecture is complex and the number of power-domains used in the design is high, selective reporting of a part of design is needed The lack of an industry standard in this regard resulted in inconsistency in the different ad-hoc approaches adopted by different tool vendors.

To keep pace with the increasing complexity of low-power architectures the 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 how the UPF 3.0 information model HDL package functions and Tcl query functions can be used to do innovative things, which are often a very important low-power design verification criteria. In this paper we will present some innovative ways of writing PA apps using the UPF 3.0 information model HDL package functions and Tcl query functions. The paper also demonstrates how these low-power applications (aka PA apps) can help in reporting, debugging and self-checking of low-power designs. We will also highlight how these apps will help offer an efficient way to significantly save verification effort and time.

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.

- Hard macro: A block that has been completely implemented and can be used as it is in other blocks. This can be modeled by an hardware description language (HDL) module for verification or as a library cell for implementation

II. UPF 3.0 INFORMATION MODEL

UPF 3.0 has come up with the concept of an information model to represent the low-power objects and concepts in a structured and consistent manner. This information model captures the low-power management information. This is the result of application of low-power UPF commands on the designs. It consists a set of objects and various information-bearing properties defined for those objects. It also defines the relationship between the HDL and UPF. It provides a set of well-defined APIs to query the low-power information in either Tcl or in HDL.

UPF 3.0 information model Tcl APIs can be used to query the static information of a low-power object, e.g. file/line detail of a UPF object or a list of isolation strategies of a power domain and other similar things. To get the dynamic information, we can rely on Tcl APIs provided by the verification tools (simulators) to access the dynamic values of the UPF and RTL objects. Together with the static and dynamic information, innovative applications can be written to help with the checking and debugging of the design.

UPF 3.0 also presents the HDL package functions and native HDL object definition for the UPF object which has some dynamic information, e.g. power domain, power states, etc. Native object definition and usage has been given in the example in the following section. Using these HDL package functions the user can access the static and dynamic information of low-power objects in HDL. This capability can be leveraged to help verification engineers create random verification scenarios.

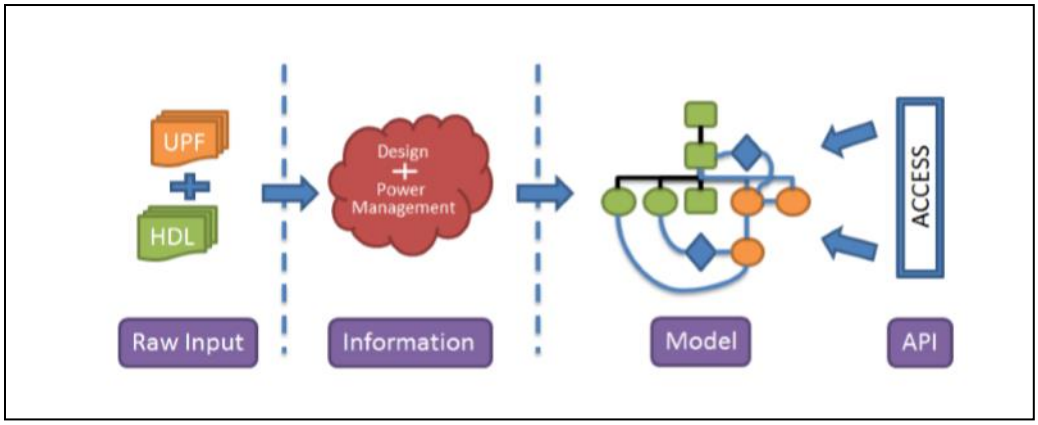

Figure 1

III. KEY COMPONENTS OF THE UPF 3.0 INFORMATION MODEL

There are two main components of the information model.

A. Objects:

These are the primary holders of information, accessed by handle ID. They represent UPF, HDL and the relationship between them. There are three main classes of objects, namely:

- UPF Objects: Model objects created by UPF.
- HDL Objects: Model objects representing the HDL design.
- Relationship Objects: Objects that model the relationship of UPF and HDL objects, e.g. upfExtentT, upfCellInfoT.

B. Properties:

These are the basic pieces of information, accessed by property ID, such as UPF_NAME, UPF_ISOLATION_STRATEGIES.

IV. UPF 3.0 HDL PACKAGE FUNCTIONS

A. Native HDL representation

UPF 3.0 defines the native HDL representation for the objects that have 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 { upfHandleT handle; **upfPowerStateObjT current_state;** } upfPdSsObjT
upfPowerStateObjT	struct { upfHandleT handle; **upfBooleanT is_active;** } upfPowerStateObjT
upfBooleanObjT	struct { upfHandleT handle; **upfBooleanT current_value;** } upfBooleanObjT
upfSupplyObjT	struct { upfHandleT handle; **upfSupplyTypeT current_value;** } upfSupplyObjT

In Table 1 above, the field representing the dynamic property of the object has been highlighted in bold. For example, 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 the 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.

The 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 that are used to access the low-power objects and their properties. These are broadly classified in the following five different classes of functions.

1. **HDL access functions**: These are the basic functions to access the low-power objects and properties. For example, the 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 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 the UPF 3.0 information model allow read access to its properties. In the case of dynamic properties these functions return the current dynamic value/state of that property when this function is called, for example:

```
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 manipulation of low-power objects from testbench or simulation model. For example, `supply_on("supply_net_name", value)`. The following objects allow immediate write access:
    a. upfPowerStateT
    b. upfLogicNetT
    c. upfLogicPortT

d. upfSupplyNetT
e. upfSupplyPortT

These functions are a powerful tool for users to manipulate low-power objects during simulation from a testbench.

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

```
upfSupplyObjT vdd_monitor;
upf_create_object_mirror("/top/dut_i/vdd", "vdd_monitor");
```

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

V. UPF 3.0 TCL APIS

The UPF 3.0 information model defines a number of Tcl query command to access the low-power objects and properties. UPF 3.0 introduced a Tcl-based Information Model Application Programmable Interface (API). These APIs can be used to access PA information:

Basic Tcl APIs

To get various attributes on a given object

```
upf_query_object_attributes obj –attribute <attr_name> -detailed
```

To get the type of the object

```
upf_query_object_type obj
```

```
To check if an object belongs to a particular group
upf_object_in_group obj –group <group_id>
```

```
To get the full hier path of an object relative to given scope
upf_query_object_pathname obj –relative_to <object_handle>
```

Example

```
upf_query_object_properties /tb/top/pd.iso_strategy -property upf_clamp_value
```

An object handle is used to access any power aware information. A handle can be a pathname, e.g. /tb/top1/PD1.ret1, or some tool assigned ID, e.g. #UPFEXTENT1234#.

A. Building Tcl Based Low-Power Apps using Tcl APIs

Tcl based apps are nothing but Tcl procedures that users can write for special requirements, such as reporting, debugging or checking of the design. Building blocks of Tcl procs (Tcl Low-Power Apps) include:

- UPF 3.0 has four basic APIs which can be used to access any UPF information.
- Tcl APIs provided by verification tools (simulators) to access the dynamic data.

Once an app is built using the above APIs, it can be run either in a verification tool environment, at their static time, to get static information or post sim to get both static and dynamic waveform data. The following is an example on how the user can build an app to find the source of corruption/retention of a signal and see the values of these signals.

UPF:

```
set_scope /tb/chip_top
create_power_domain PD_CAMERA        -include_scope
```

```
create_supply_net   pd_pwr    -domain PD_CAMERA
create_supply_set ss -function {power pd_pwr} -function {ground G_pd_net}
associate_supply_set ss -handle PD_CAMERA.primary
```

Here signal in question is /tb/chip_top/c which is corrupted at some time instance in simulation. The goal is to find the source of corruption of this signal.

```
#aliasing upf_query_object_properties to simple name such as alias
alias query upf_query_object_properties
```

 # Step 1: Get the properties of the signal

```
examine tb/chip_top/c
   # 1'bx
query tb/chip_top/c
   # { {upf_name c} {upf_parent /tb/chip_top} {upf_cell_info #UPFCELL0_71653#}
{upf_port_dir UPF_DIR_OUT} }
```

 # Step 2: Get the properties of cell applied on that signal

```
query #UPFCELL0_71653#
   # {{upf_cell_kind upf_cell_corrupt} {upf_hdl_cell_kind upf_hdlcell_comb}
{upf_cell_origin upf_origin_inferred} {upf_source_extents {#UPFEXTENT2130711#}} }
```

 # Step 3: Get the properties on source extent (extent of power domain, retention strategy etc.) of the cell

```
query #UPFEXTENT2130711#
   # { {upf_hdl_element tb/chip_top} {upf_object tb/chip_top/PD_CAMERA /*power
domain*/} }
```

 # Step 4: Get the supplies of the upf_object (power domain, retention strategy etc.)

```
query /tb/chip_top/PD_CAMERA -property upf_supply_set_handles
      # {/tb/chip_top/PD_CAMERA.primary /tb/chip_top/PD_CAMERA.default_retention
/tb/chip_top/PD_CAMERA.default_isolation}
```

 # Step 5: Get the power (or other relevant function) of the primary supply set

```
query /tb/chip_top/PD_CAMERA.primary.power
   # { {upf_name power} {upf_creation_scope /tb/chip_top/PD_CAMERA} {upf_parent
/tb/chip_top/PD_CAMERA.primary} {upf_ref_kind upf_ref_power} {upf_ref_object
/tb/chip_top/pd_pwr} }
```

 # Step 6: Check the value of UPF supply net

```
examine tb/chip_top/pd_pwr
   # OFF OV
```

VI. EXAMPLES AND CASE STUDIES

When using Tcl APIs and HDL package functions a number of novel objectives can be achieved. This section captures some of the innovative low-power apps based on information model APIs to solve practical low-power verification problems, which otherwise are relatively difficult to solve and users have to rely on tool vendors for those specific features. The paper captures a few useful applications. However, along similar lines, users can write their own application for various needs.

A. Low-Power Apps based on HDL Package Functions

Low-Power App 1: (Coverage App) Coverage of a low-power design using HDL Package Functions

In a low-power design, it is of utmost importance for a verification engineer to ensure that all IPs in the design behave properly in OFF/ON mode. They also need to ensure that transitions from ON->OFF and OFF->ON have also been verified. This requirement can be achieved by creating a coverage infrastructure to ensure the full coverage of the simstate property of the primary supply set of all power domains.

The aim of this application is to do simstate coverage (Normal/Corrupt) of all the powerdomains in the design. The application will cover the NORMAL-> CORRUPT and CORRUPT->NORMAL transitions for each power domain in the design. We have presented below how UPF 3.0 HDL package functions can be used to achieve this.

Step1: Mirror UPF objects to HDL objects

```
// Native HDL representation for power domains
typedef struct {
   upfHandleT handle;
   upfSimstateT simstates;
} upfPdObjT;
```

Use the mirror function to continuously monitor the simstate of all the power domain in the design

```
      pd_iter = upf_get_all_power_domains();
      pd_hndl = upf_iter_get_next(pd_iter);
      while (pd_hndl) begin
          pd_obj = "power_domain_objs[";
          pd_cnt_str.itoa(pd_cnt);
          pd_obj = {pd_obj, pd_cnt_str};
          pd_obj = {pd_obj, "]"};
          upf_create_object_mirror (upf_query_object_pathname(pd_hndl), pd_obj);
          pd_cnt++;
          pd_hndl = upf_iter_get_next(pd_iter);
      end
```

Step 2: Covergroup definition for state and transition coverage

```
      covergroup PD_STATE_COVERAGE (string pd_name, ref upfSimstateE simstate)
@( simstate);
      CORRUPT: coverpoint simstate
                    { bins ACTIVE = {CORRUPT}; }
      NORMAL: coverpoint simstate
                    { bins ACTIVE = {NORMAL}; }
      COA: coverpoint simstate
                    { bins ACTIVE = {CORRUPT_ON_ACTIVITY}; }
      option.per_instance = 1;
      type_option.merge_instances = 0;
      option.comment = pd_name;
      endgroup

      covergroup PD_TRANS_COVERAGE (string pd_name, ref upfSimstateE simstate)
@( simstate);
      TRANSITION_COVERAGE:coverpoint simstate
      {
             bins OFF_to_ON = (CORRUPT => NORMAL);
             bins ON_to_OFF = (NORMAL => CORRUPT);
             bins ON_COA_OFF = (NORMAL => CORRUPT_ON_ACTIVITY => CORRUPT);
      }
      option.per_instance = 1;M
      type_option.merge_instances = 0;
      option.comment = pd_name;
      endgroup
```

Step 3: Instantiation of coverage module:

```
      PD_STATE_COVERAGE pd_state_cov [$];
      PD_TRANS_COVERAGE pd_trans_cov [$];
      initial begin
          for (int i = 0; i < pd_cnt; i++) begin
            pd_state_cov[i] = new
```

```
                (upf_query_object_pathname(power_domain_objs[i].handle),
                power_domain_objs[i].simstate);
          pd_trans_cov[i] = new
                (upf_query_object_pathname(power_domain_objs[i].handle),
                power_domain_objs[i].simstate);
        end
    end
```

Monitor the simstates of a power domain: User can also monitor the simstates of one or more power domains of interest.

```
always @(power_domain_objs[0].simstate) begin
    $display ($time, "%s Power Domain '%s' simstate changed to '%s'", identstr,
upf_query_object_pathname(power_domain_objs[0].handle),
get_simstate_str(power_domain_objs[0].simstate));
end
```

Low-Power App 2: Write function to print current simstates of a power domain using HDL Package Functions

User can write following set of functions to print the simstates of all the power domains of the design at any instance of time in simulation.

```
function string get_simstate_str(power_state_simstate simState);
if(simState == NORMAL)
    get_simstate_str = "NORMAL";
if(simState == CORRUPT)
    get_simstate_str = "CORRUPT";
else if(simState == CORRUPT_ON_ACTIVITY)
    get_simstate_str = "CORRUPT_ON_ACTIVITY";
else if(simState == CORRUPT_STATE_ON_ACTIVITY)
    get_simstate_str = "CORRUPT_STATE_ON_ACTIVITY";
else if(simState == CORRUPT_STATE_ON_CHANGE)
    get_simstate_str = "CORRUPT_STATE_ON_CHANGE";
else if(simState == CORRUPT_ON_CHANGE)
    get_simstate_str = "CORRUPT_ON_CHANGE";
endfunction

function reg print_current_state_of_hndl(upfHandleT hndl);
    upfHandleT state_hndl, simstates_hndl, pd_nm_hndl, state_nm_hndl;
    upfHandleT line_no_hndl, file_nm_hndl, iter_hndl;
    int simstate;

    state_hndl = upf_query_object_properties(hndl, UPF_CURRENT_STATE);
    pd_nm_hndl = upf_query_object_properties(hndl, UPF_NAME);
    file_nm_hndl = upf_query_object_properties(hndl, UPF_FILE);
    line_no_hndl = upf_query_object_properties(hndl, UPF_LINE);
    state_nm_hndl = upf_query_object_properties(state_hndl, UPF_NAME);
    simstate_hndl = upf_query_object_properties(hndl, UPF_SIMSTATE);
    simstate = upf_get_value_int(simstate_hndl);

    $display ($time, "%s Power domain: %s (%s:%0d), Current simstate: %s",
identstr,
        upf_get_value_str(pd_nm_hndl), upf_get_value_str(file_nm_hndl),
upf_get_value_int(line_no_hndl), get_simstate_str(upfSimstateE'(simstate)) );
    return 1;
endfunction

function reg print_pd_simstates();
    upfHandleT pd_iter;
    upfHandleT pd_hndl;
    int pd_cnt;
    pd_iter = upf_get_all_power_domains();
    pd_hndl = upf_iter_get_next(pd_iter);
```

```
    while (pd_hndl) begin
        print_current_state_of_hndl(pd_hndl);
        power_domains[pd_cnt++] = pd_hndl;
        pd_hndl = upf_iter_get_next(pd_iter);
    end
    return 1;
endfunction
```

B. Low-Power Apps based on Tcl APIs

Low-Power App 3: (Reporting App) UPF query_* commands

Reporting is an essential part of the low-power verification process. Once the power intent is captured in a UPF file, it is important for the verification and design engineers to know that it has be captured as the original intention. This requirement can be fulfilled by query_* procs. These query commands can query the UPF data as interpreted by the verification tools and stored in the information model. The output of query commands can be used to do selective reporting.

```
interp alias {} query {} upf_query_object_properties;
interp alias {} type {} upf_query_object_type;
interp alias {} group {} upf_object_in_class;
interp alias {} name {} upf_query_object_pathname

proc query_port_direction {{port_name ""} args} {
   set direction [query $port_name -property upf_port_dir]
   switch $direction {
      UPF_DIR_IN {set result "in"}
      UPF_DIR_OUT {set result "out"}
      UPF_DIR_INOUT {set result "inout"}
      default { set result ""}
   }
   return $result
}
```

Usage:

```
query_port_direction /tb/t/a/vdd
```

Result: “in”

```
proc query_power_domain {{domain_name} args} {
   if {[type $ domain_name] == "upfPowerDomainT"} {
      set property [query -verbose $ domain_name]
      set element ""
      set extents [lindex [lindex $property 5] 1]
      foreach i $extents {
            set l [split [string map [list "(" \0] $i] \0 ]
            lappend element [string trimright [lindex $l 1] ")"]
      }
      #lappend result "domain_name [lindex [lindex $property 0] 1]"
      puts "{domain_name: [lindex [lindex $property 0] 1]}"
      puts "{scope: [lindex [lindex $property 3] 1]}"
      puts "{supply: [lindex [lindex $property 6] 1]}"
      puts "{power_switch: [lindex [lindex $property 12] 1]}"
      puts "{pd_states: [lindex [lindex $property 13] 1]}"
      puts "{elements: $element}"
      #return $result
   } else {
      return "ERROR : Invalid arguments. arg '$ domain_name' not a
'Power_Domain'"
   }
}
```

Usage:

```
query_power_domain /tb/pd
```

Result:
{domain_name: pd}
{scope: /tb}
{supply: /tb/pd.primary /tb/pd.default_retention /tb/pd.default_isolation}
{power_switch: /tb/pd_sw}
{pd_states: /tb/pd.ON /tb/pd.SLEEP}
{elements: /tb/top2/m4 /tb/top2/m4/iso_inst1}

Low-Power App 4: (Debug/Reporting App) Get all attribute information

In a low-power design, along with the UPF file, some of the power intent can be present in a Liberty file as well. The Liberty information is annotated on RTL objects using attributes which can then be further updated using the UPF command `set_port_attributes`. In a low-power design containing hard macros, attribute information plays a vital role when debugging or reporting. These low-power attributes can be present on an instance or port of an instance. This low-power app can be used on any signal or instance in the design to get the attribute information and the respective signal values if wave data is available.

```
proc pa_query_attributes {{object} args} {
   set result ""
   if {[type $object_name]== "upfHdlScopeT"} {
     lappend result "model [lindex [query $object -property upf_model_name] 0]"
     lappend result "file [lindex [query $object -property upf_model_name] 1]"
     lappend result "line [lindex [query $object -property upf_model_name] 2]"
   } elseif {[type $object_name]== "upfHdlPortBitT"} {
      set parent [query $object_name -property upf_parent]
      lappend result "parent_model [lindex [query $parent -property
upf_model_name] 0]"
   } else {
      return "ERROR : Invalid object. Expecting 'HdlPort' or 'Instance'"
   }
   set attr [query $object_name -property upf_hdl_attributes]
   if {$attr != ""} {
      lappend result "attributes $attr"
   } else {
      lappend result "attributes NO_ATTRIBUTE_SET"
   }
   #printing result
   foreach i $result {
      if { [lindex $i 0] != "attributes"} {
            puts "{$i}"
      } else {
            puts "\{[lindex $i 0]"
            set i [lreplace $i 0 0]
            foreach  j $i {
                  puts "\t> {$j}"
            }
            puts "\}"
      }
   }
   return $result
}
```

Usage:

```
   pa_query_attributes tb/dut/ab5
```

Result:

```
   {model analog}
   {file analog.sv} {line 53}
   {attributes
           > {mspa_cell_functionality pa {analog.lib} {28}}
```

```
            > {level_shifter_type HL {analog.lib} {28}}
            > {is_level_shifter true {analog.lib} {28}}
    }
```

Low-Power App 5: (Debugging App) Trace drivers of UPF objects

For a low-power design consisting of RTL along with UPF, all the supply network including creation of port, nets and their connection is written inside the UPF file. Debugging of the supply network is a major problem that many verification engineers come across. This low-power app is useful as it can trace the driver of any UPF objects along with printing the values of all the ports and nets in the path. The input of this app can be either a UPF created supply, Liberty created supply pin or a supply defined in HDL.

```
proc pa_query_drivers {{object} args} {
    set fanin $object
    set driver ""
    append driver $object
    while {[query $fanin -property upf_fanin_conn] != ""} {
        set driver [concat $driver "[examine $fanin] <-"]
        if { [llength [query $fanin -property upf_fanin_conn]] > 1 } {
               set resolution [query $fanin -property upf_resolve_type]
               set fanin [query $fanin -property upf_fanin_conn]
               foreach index $fanin {
                      set driver [concat $driver "$index [examine $index]"]
               }
               set driver [concat $driver "\{$resolution\}"]
               break
        }
        set driver [concat $driver "[query $fanin -property upf_fanin_conn]"]
        set fanin [query $fanin -property upf_fanin_conn]
    }
    if {[llength $fanin] < 2 } {
        set driver [concat $driver "[examine $fanin]"]
    }
    return $driver
}
```

Usage:

```
pa_query_drivers /tb/t1/m1/b1/vd_bot
```

Result:

```
    /tb/t1/m1/b1/vd_bot {OFF 0} <- /tb/t1/m1/b1/vport1_bot {OFF 0}
        <- /tb/t1/m1/vd_mid {OFF 0} <- /tb/t1/m1/vport1_mid {OFF 0}
            <- /tb/t1/vd_top {OFF 0} <- /tb/t1/vport2_top {OFF 0} /tb/t1/vport1_top
{OFF 0} {PARALLEL}
```

VII. POSSIBLE USAGE OF HDL PACKAGE FUNCTIONS AND TCL APPS

As observed in the above sections, there are two main approaches to access and manipulate the low-power objects and properties. One is HDL package functions and the other is to use the Tcl query commands. There are different scenarios in which one or the other approach is suited. Following table summarizes the various usage scenarios where HDL package functions or Tcl query commands can be used.

Table 3.**HDL Package functions**	**Tcl Apps**
Useful for coverage of low-power objects	Useful in selective reporting
Useful for transition coverage of power states	Batch mode debug (live sim or post sim)
Directed assertions on low-power objects (e.g. simstates of power domain)	Power aware static checking
Dynamic checks involving lower-power objects	

VIII. BENEFITS OVER CONVENTIONAL APPROACHES

Verifications engineers can use the proposed verification approach to achieve early low-power verification closure. The approach mentioned in this paper using the UPF 3.0 information model provides a number of benefits. This approach is consistent across tool vendors as it is based on the UPF 3.0 standard. The learning curve for the users is not steep. Also the user scripts created to use the proposed solution are easily scalable to bigger and more complex design scenarios.

IX. CONCLUSION

The low power designs today are incredibly complex with intricate power architecture. A thorough low-power verification is a must for such designs, as any power bug left can cause a huge setback. In this paper we have discussed the challenges with the current low-power verification method and how those challenge can be addressed better with UPF 3.0. We discussed the concepts of UPF 3.0 information model and APIs to represent and access the lower power information which is the result of application of UPF on the design. We also presented with examples and case studies how UPF 3.0 information model concepts can be used to develop a more consistent, robust and scalable low-power verification platform. In the end we discussed the benefits of using the proposed approach over conventional approaches.

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] “Awashesh Kumar, Madhur Bhargava”, Random Directed Low Power Coverage Methodology: A Smart Approach to Power Aware Verification Closure, DVCon USA 2017
[3] “Awashesh Kumar, Madhur Bhargava”, Unleashing the Power of UPF 3.0: An innovative approach for faster and robust Low-power coverage, DVCon India 2017