2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Probing UPF Dynamic Objects:

Methodologies to Build Your Custom *Low*-Power Verification Platform

Progyna Khondkar, Product Engineering, Mentor a Siemens Business, Austin, Texas, USA
(*progyna_khondkar@mentor.com*)

***Abstract*—Oftentimes low-power design and verification engineers crave for ways to continuously probe ON, OFF status of a power domains, or different states of a supply sets, supply nets, supply ports, logic ports, power switch acknowledge ports, or conditions of different strategies (e.g. isolation, retention), or discrete changes in supply voltages or even wants to populate cover-bins with coverage data from unconventional power state transitions. Apparently there were no ways to continuously monitor the dynamic properties of UPF objects –like, just noted above – let's say 'the current state of a strategy' and utilize the information to develop custom low-power verification environment. This paper proposes a completely new low-power verification methodology on the key concepts of low-power (UPF) information model (UPFIM) [2] that directly imply Tcl and SV HDL API to user low-power designs. The novel methodology allows to query any dynamic properties of UPF objects – like continuously probe ON, OFF status of a power domains (during elaboration steps) through Tcl API and passed the objects information on to appropriately instantiated SV API based design codes. For example, Tcl API can be used on the simulation execution fly to populate any attributes for low-power SystemVerilog checker modules that are already quarried and bound during elaboration steps into RTL design through UPF bind_checker.**

Keywords—UPF; low-power designs; Information Model; dynamic objects; bind_checker; custom low-power checker

I. INTRODUCTION

The IEEE 1801-2015 (UPF 3.0 and later updated in 1801-2018/UPF 3.1) standard that specify the low-power intent for any design, introduced a concept of low-power (UPF) information model (UPFIM) which can be useful to address these complex verification challenges. This paper proposes a completely new low-power verification methodology on the key concepts of UPFIM that directly imply Tcl and SV HDL API to user low-power design. The novel methodology allows to query any dynamic properties of UPF objects (during elaboration steps) through Tcl API and passed on the objects information on to appropriately instantiated SV API based design codes. For example, Tcl API can be used on the simulation execution fly to populate any attributes for low-power SystemVerilog checker modules that are already quarried and bound (during elaboration steps) into RTL design through UPF **bind_checker**.

Evidently such functionalities were impossible without the proposed new methodology where user on the first iteration of simulation flow - requires to find out the appropriate properties of the UPF objects (e.g. current state of save-restore condition of retention strategies associated with a power domain) from UPFIM database (UPFIMDB) through Tcl quarries, then populate the compatible HDL port types with the value and rerun the simulation to get the ultimate benefits of UPFIM- as well the results of custom checker assertions. In order to fully comprehend the proposed methodology and exploit the merits in any design, it is important to at least grasp the gist of UPF, UPFIM, Phases of UPF processing and UPF **bind_checker**.

UPF: The Unified Power Format (UPF) also known as IEEE-1801, is not just a language to denote low-power intents or power specifications for a design – it's a complete command set for verifying such low-power designs. The UPF is the ultimate abstraction of low-power methodologies today. It provides the concepts and the artifacts of power management architecture, power aware verification and low-power implementation for any design. It provides the notions of power architecture from very early stage of design abstraction. Now it's the industry trend and standard for lowering static and dynamic power dissipation in every digital design. Overwhelmingly UPF standout as the only alternative choice of lowering power dissipation when fabrication process technology advanced below 65nm.

UPF Low-Power Design: When UPF is developed for any design, it overlaid and architects the low-power artifacts on it. For examples, UPF allows to confines different design (hierarchical instances) portions in different power domains, provides power states, supply sets (nets, ports) and simulation status, implements power

management cells (like isolator, power switches, retention flops, level-shifters, repeaters etc.). So UPF is not just a verification or power management methodology, it's the ultimate key to instrument any design to make it physically low-power.

UPFIM: The UPF information model captures the power-management information from UPF semantics applied on a design. The model contains information about UPF objects (e.g. power domain, supply sets, strategies, design groups, models, instances, etc.) and design in order to comprehensively capture the power intent in a standard database (UPFIMDB) format, that can be queried via UPF (Tcl) queries and native representations of UPF HDL package functions, calls API.

UPF Processing Phases: There are certain phases before the UPF objects or its implication on a design can be quarried and/or verified (simulated). UPF LRM defines abstract of UPF processing phases as follows:

List 1: UPF Processing Phases on Designs

Phase 1 Read UPF specifications
Phase 2 Build UPF model
Phase 3 Recognize the implemented UPF and process simulation controls
Phase 4 Apply UPF model to the design, including any checkers introduced by the UPF **bind_checker**
Phase 5 Query UPF model through Tcl and HDL API based quarry. This phase also implements checkers quarries resulting from the **bind_checker**.

UPF bind_checker: UPF provides a powerful mechanism to define a custom low-power checker or assertion within a layer that completely separate it from user design HDL codes. This is done by embedding the binding of the design and checker within the UPF/Tcl file through the UPF **bind_checker** command. As a result, it provides a consolidated verification mechanism and allows simulator to access all instances of a target design with a custom checker within the current scope. The **bind_checker** assertions are distinctively different from SystemVerilog assertions in that they can access all the UPF objects – i.e. UPF power supply, power states etc. through use of Tcl query commands or HDL native functional package API.

A. Motivation and Contribution of this Paper:

Our core objective is to pave the way to continuously probe UPF objects and utilize them in productive manner in making custom low-power verification platform. Since availability of the dynamic properties of UPF objects to design and verification tools (simulator) are strictly governed by UPF LRM processing phases, we proposed a methodology that orchestrates the processing of UPF phases with simulator steps. In this paper we conducted empirical research to identify and establish a complete user perception that allows to query UPFIMDB on the simulation execution fly and populate any attributes for low-power SystemVerilog checker modules that are bound during simulation elaboration steps into RTL design through UPF **bind_checker**.

B. Organization of this Paper:

This paper is organized in the following structure. Section I introduces the problem formulation, key concepts of UPF fundamentals and relevant UPF topics in conjunction to the proposed methodology. Section II explains the specific detail of the proposal. The implementation detail and outcome of the proposed methodology are shown in section III. The final sections IV draws the conclusion. The references are shown at the end.

II. UNDERSTANDING THE PROPOSED METHODOLOGY

As noted in previous section, the latest UPF 3.1 or IEEE1801 LRM [2] provides two APIs for accessing the UPFIM, namely the Tcl and the native SV HDL API. Also as defined in "UPF Processing Phases", the simulation related controls are enabled during phase 3 to 5, which are the elaboration step of a general purpose or low-power three steps simulator. To note, these three steps of simulators are, compilation of HDL codes of the design, design elaboration & optimization (UPF/Low-power instrumentation on design hierarchies) and execution of simulation (for example, the steps are vlog/vcom, vopt and vsim for Questa™ Simulator).

Precisely, simulators creates the UPFIM database (UPFIMDB) at the end of optimization step (i.e. vopt). Obviously, accessing this database can only be possible by the execution steps (i.e. vsim) of such simulators. Unfortunately this poses certain limitations on custom verification productivity like checker based monitoring of status (ON, OFF) of a power domains, or different states of a supply sets, supply nets, supply ports, logic ports, power switch acknowledge ports, or conditions of different strategies (e.g. isolation, retention), or discrete changes in supply voltages or populating cover-bins with coverage data from unconventional power state transitions. Specifically following verification model are largely affected by the limitations to access UPFIMDB.

List 2: Limiting factors of Accessing UPFIMDB

1. Use of Tcl API to get information from UPFIMDB that can be passed on to appropriately instantiated SV API based code
2. Use of low-power checker modules that are bound into a design using UPF **bind_checker** command where Tcl API is used to query attributes on the fly, that are passed to the appropriate checker instances, created by the **bind_checker** command

Our goal is to handle the UPF processing phases (phase 1 through 5) at the design elaboration & optimization (i.e. vopt) steps in such a way that simulator leverages a consolidated UPF flow that consist of UPF commands and UPFIM Tcl API in single UPF file, and allows passing the UPFIM property via ports of native HDL representation to checker module in a **bind_checker** setup. It's important to understand that, integration and implementation of such verification mechanism in verification tools (simulator) must be UPF LRM [2] compliant. Such LRM guidelines are shown below.

List 3: UPF LRM Guidelines for Processing Phases

1. API those query the UPFIM (i.e., UPF queries and UPF HDL package functions) will only works after Phase 4 of UPF processing.
2. The UPFIM shall not capture any intermediate steps involved in reaching phase 4.
3. It shall be an error if a query function appears in a power model
4. It shall be an error if a query function is followed by a UPF command that would affect the power intent. This implies that query functions should be the last set of code processed when using a consolidated UPF flow.

The query functions of UPFIM that allows to probe dynamics objects are listed below.

List 4: Basic Query Functions from UPFIM

1. **upf_query_object_properties** – This allows to query properties on a given object.
2. **upf_query_object_pathname** – This is a helper query command that is used to return the hierarchical pathname relative to given scope.
3. **upf_query_object_type** – This allows to query the type of a given object.
4. **upf_object_in_class** – This functions checks if an object belongs to a particular class and allows error checking mechanism in the proposed verification methodology.

III. PROBING DYNAMIC OBJECTS

Based on our discussion in section II, now it's distinctive that our proposed methodology utilize the foundation of UPFIM in accordance to UPF processing phases. This ultimately accommodate custom low-power checker query processing in consolidated manner. In the proposed use model, the Tcl API query to UPFIMDB is used together with **bind_checker** to bind the checker whose interface use the corresponding SV HDL native representation types. The methodology automatically creates object of the correct type during design & UPF elaboration steps of a three step simulation flow and allows continuous access to reflect it on the port(s) of the SV checker module. Let us explain the methodology based on following retention cheeker examples. In this example, the checker will fire success assertion while save and restore operations are performed in correct manner. Alternatively flags violation when save and/or restore operations fails.

Table 1: Custom Retention (Save-Restore Correct Operation) Checker Example

```
## Custom SV Checker 'ret_checker.sv'
import UPF::*;
module checker_retention(sav_sig, res_sig, sav_cond, res_cond, ret_clk);
    input sav_sig, res_sig;
    input upfExpressionT sav_cond;
    input upfExpressionT res_cond;
    input ret_clk;
    wire clk;
    assign #1step clk = ret_clk;

    property p1;
        @(posedge clk) (sav_sig |-> sav_cond.current_value);
    endproperty

    property p2;
        @(posedge clk) (!res_sig |-> res_cond.current_value);
    endproperty
```

```
    assert property (p1) $display(">>>%t ----- Save success", $time());
        else $display(">>>%t ----- Save not executed", $time());

    assert property (p2) $display(">>>%t ----- Restore success", $time());
        else $display(">>>%t ----- Restore not executed", $time());
endmodule
```

Table 2: Regular UPF bind and Query Functions for Custom Retention Checker Example

```
## Regular Power Management UPF 'dut_top.upf'
create_power domaon PD -elements {<list of elements e.g.> top_vl top_vl1}
... ... ...
## Retention Strategy
46 set_retention pd_retention \
47        -domain pd \
48        -retention_supply ss \
49        -elements { top_vl1/q top_vh1/q } \
50        -save_signal { ret1 posedge } \
51        -restore_signal { ret1 negedge } \
52        -save_condition  {!UPF_GENERIC_CLOCK && sc} \
53        -restore_condition {UPF_GENERIC_CLOCK && !UPF_GENERIC_ASYNC_LOAD && rc}
## Regular UPF Commands/Options
## For e.g. defining other PD, PD's supply set, association,
## Power state etc.

## Binding design module, Regular UPF and Custom Retention Checker through UPF bind_checker
bind_checker $instance_name -module checker_retention -bind_to tb -ports $ports_list
```

```
## Utilization of UPF Tcl Query Functions in Consolidated UPF flow
foreach RET_STRATEGY [upf_query_object_properties $PD -property upf_retention_strategies]
 {set ret_save_signal [upf_query_object_properties $RET_STRATEGY -property upf_save_signal]
  set ret_sav_sig_port [list sav_sig [upf_query_object_properties \
      $ret_save_signal -property upf_control_signal]]
  set ret_restore_signal [upf_query_object_properties $RET_STRATEGY -property upf_restore_signal]
  set ret_res_sig_port [list res_sig [upf_query_object_properties \
      $ret_restore_signal -property upf_control_signal]]
  set ret_save_condition [upf_query_object_properties $RET_STRATEGY -property upf_save_condition]
  set ret_sav_cond_port [list sav_cond $ret_save_condition]
  set ret_restore_condition [upf_query_object_properties $RET_STRATEGY -property upf_restore_condition]
  set ret_res_cond_port [list res_cond $ret_restore_condition]
  set RET_STRATEGY_NAME [upf_query_object_properties $RET_STRATEGY -property upf_name]
  set RET_QUERY [query_retention $RET_STRATEGY_NAME -domain $PD -detailed]
        array set RET_DETAILS [join $RET_QUERY]
        set RET_CLK $RET_DETAILS(upf_generic_clock)
        set ret_clk_port [list ret_clk $RET_CLK]
        set ports_list {} lappend ports_list $ret_sav_sig_port $ret_res_sig_port $ret_sav_cond_port
        $ret_res_cond_port $ret_clk_port
        set ret_path [upf_query_object_pathname $RET_STRATEGY]
        set is_ret [upf_object_in_class $RET_STRATEGY -class upfRetentionStrategyT]
        set cell_type [upf_query_object_type $RET_STRATEGY]
## Printing Useful Information for the Retention Strategy
puts "RET_INFO_STRATEGY: $RET_STRATEGY\n PATH: $ret_path\n type: $cell_type\n"}
```

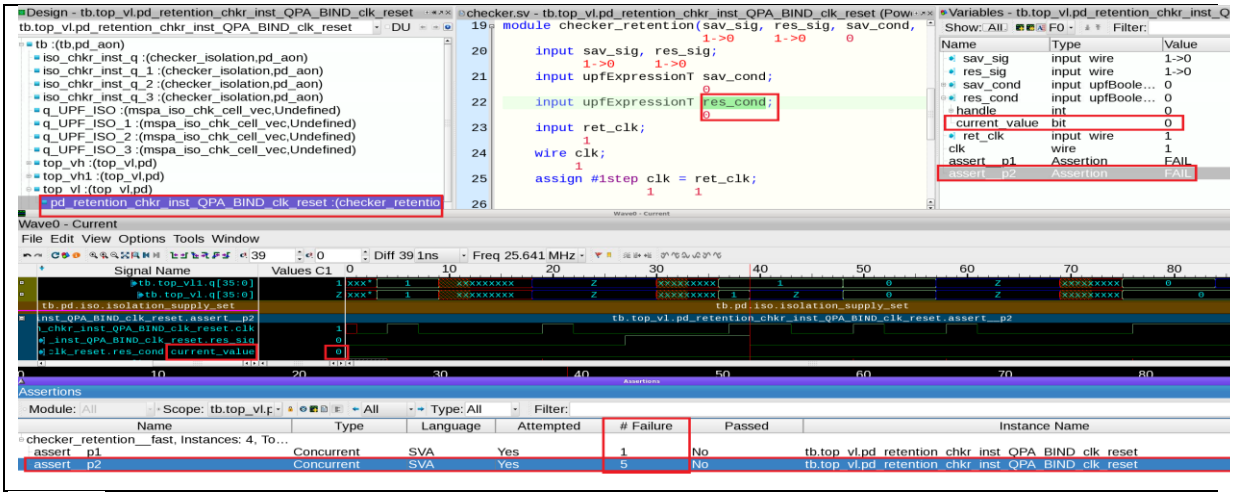
Table 3: Transcript Results for Custom Retention Checker Example

```
## Useful Information for the Retention Strategy
-- Loading module checker_retention
STRATEGY: /tb/pd.pd_retention1
PATH: /tb/pd.pd_retention1
type: upfRetentionStrategyT
... ... ...
## Assert Property p1 and p2
# ** Error: (vsim-8906) QPA_RET_SEQ_ACT: Time: 36 ns,  clock toggled  during retention period for retention
element(s) in scope '/tb/top_vl': q
# File: ~/test.upf, Line:46, Power Domain:/tb/pd
#                   37 pwr= 1, ret= 1, ret1= 0, clk= 1, rst= 0, d=1, q_vl= 01 01, q_vl1= 01
#
# >>>                  37 ----- Save not executed
# >>>                  37 ----- Restore success
```

The use model shows the ultimate objective of consolidating phase 1~5, in such a way that it allows a single UPF flow containing both UPF (Tcl) and UPFIM Tcl API integrated with **bind_checker** commands during elaboration step. As a consequence, the execution in simulation step will reveal the final results to its entirety at once. Please note that, the object passing between Tcl query and checker port may or may not contain UPF native HDL representation type, hence *upfExpressionT* appears only in checker but not in UPF Tcl query. Because, when ports of native HDL representation types are defined in the checker model, then the elaboration step will

automatically create a connection of continuous access using continuous mirroring API defined in the UPFIM. Following figure 1 shows the implemented results of proposed methodology in third step of design & UPF execution (i.e. vsim). The current value of restore condition, property 'p2 pass/fail' status etc. can be evaluated through the verification platform captured below.

Figure 1 Custom Checker Verification Platform



Hence it's evident that propose methodology allows to create checker module in such a way where user can readily utilize object passing by query function, hierarchical references as well with or without native SV HDL representations. During the implementation we recognize that UPF 3.1 LRM do not provide adequate clarification on coordinating *upfExpressionT* with retention strategy record field name spaces or UPF generics, like the UPF_GENERIC_CLOCK, UPF_GENERIC_DATA etc. (Please refer to Table 1, retention strategy save or restore condition {!UPF_GENERIC_CLOCK && sc} to comprehend the usage of record field generics). Specifically UPF mirror object function and **bind_checker** for generic expression requires further clarification from IEEE 1801. Please note that *upfExpressionT* is UPFIM class that creates relational objects to captures Boolean expressions for save, restore, retention condition, states of power switch, and states of logic and supply expressions of power state etc.

IV. Concluding Remarks

In this paper, we have proposed and implemented a novel methodology that paved the way to continuously probe UPF dynamic objects and allows to build custom low-power verification portfolio on existing low-power simulation platform. If carefully designed, these custom checkers can be reused across any low-power projects.

Acknowledgment

The author would like to express sincere gratitude to the reviewers from Mentor Graphics Corp. and TPC DVCON EU for providing useful insight and guidance in conducting the research and preparing this paper.

References

[1] Progyna Khondkar, "Low-Power Design and Power-Aware Verification", Hard Cover ISBN: 978-3-319-66618-1, October, 2017, Springer International Publishing.
[2] Design Automation Standards Committee of the IEEE Computer Society, "IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems", IEEE Std. 1801™-2018.
[3] Progyna Khondkar, et al., "How UPF 3.1 Reduces the Complexities of Reusing Power Aware Macros" March, DVCon 2020."
[4] Progyna Khondkar, et al., "Low Power Coverage: The Missing Piece in Dynamic Simulation", February March, DVCon 2018.
[5] Progyna Khondkar, et al., "Free Yourself from the Tyranny of Power State Tables with Incrementally Refinable UPF", February March, DVCon 2017.
[6] Design Automation Standards Committee of the IEEE Computer Society, "IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems", IEEE Std. 1801-2015, 5 December 2015.