Free Yourself from the Tyranny of Power State Tables with Incrementally Refinable UPF

Progyna Khondkar, Ping Yeung, Gabriel Chidolue, Joe Hupcey, Rick Koster and Madhur Bhargava
Mentor Graphics Corporation. USA.
{progyna_khondkar, ping_yeung, gabriel_chidolue, joe_hupcey, rick_koster, madhur_bhargava @mentor.com}

***Abstract*- The recent edition of IEEE 1801 specifies the power state table (PST) construct should be phased out as legacy, and instead be replaced by the new semantics of the 'add_power_state' UPF command. This paper starts with investigating the limitations of legacy PST in a complex SoC design verification environment, and how to reap the benefits of the incrementally refinable power state features through the fundamental constructs of 'add_power_state'. In order to comprehend the benefits of new semantics over PST, we perceived the new concepts of definite, indefinite and deferred power states at different design level of abstractions and connect them to demonstrate in power aware (PA) static and dynamic verification methodologies. The conceptual sets of these fundamental power states provide a framework for verification engineers and tools to comprehend refinement of power states in a successive refinement flow as information about details of the design become incrementally available. In practice we extended this conceptual set as foundation of power aware static and dynamic verification techniques. Eventually the power state concept realization also allows us to probe further into the power management components for design and IP integration in different levels of design abstractions. Our approach shows validation procedures for UPF strategies through PA static verification. With design examples and case studies we also demonstrated how to achieve PA design verification closure with state and transition coverage, as well interdependent state's cross-coverage of power domains in more flexible and controllable ways.**

I. Introduction

Since UPF was first announced in 2007 by Accellera, many of the early features- like explicit supply port, supply net and the power state table (PST)- governed UPF based low power design verification methodologies mainly from post synthesis levels and onward. However the recent update of IEEE 1801 [3] specifies intrinsic flexibility to associate a power domain with a supply set and implicate infinite ordered list of power states, augmented with incrementally refinable arguments for the objects. The objects are the rudimentary parts of the power supply architecture and the design elements, like supply sets, power domains, design groups, design models and design instances, collectively known as objects[2], [3]. The new power states definition completely replaces the conventional and tedious PST and enables multiple possible combinations of power states and their transitions models for the objects at any level of design abstraction. The incremental refinement features allow users to define new power states either on the original or on its derivatives entities for any object. Such power states provide more flexibility to represent different operational modes of a design with more controllability.

A. UPF Power States

The UPF power state is the core of the entire UPF modelling and power aware verification. The power states are specified through UPF **add_power_state** commands that define one or more power states of an object. The power states may also be represented through states of a power state tables or PSTs, which is constructed with the combination of **add_port_state**, **add_pst_state** and **create_pst** UPF commands**.** However these commands are considered as legacy in UPF 2.1 LRM and onward [1]. Because of limitations in coordinating with supply set, specifically, PST states are defined based on supply net only. And supply nets are usually available after synthesis and post-layout levels of design abstraction. In addition, there are no UPF methodologies for PST state to refine states throughout the design verification and implementation flow (DVIF), and so on. The UPF power states syntax is shown in Syntax 1, below.

Syntax 1. Snippet of UPF Power States Syntax

```
add_power_state object_name
[-supply | -domain | -group | -model | -instance] object_name
[-update] [-state {state_nameb[-supply_expr {boolean_expression}]
[-logic_expr {boolean_expression}]][-simstate simstate]
[-power_expr {power_expression}]
[-legal | -illegal]}]*
[-complete]
```

The UPF syntax dictates that the **add_power_state** attributes an object with a power state definition. The power state is specified with a state name that consists of a logic expression, supply expression and simstate. Instead of explaining the functionality of each option of the syntax, let us explain the semantics of power states through the examples of a complex SoC design shown in Figure 1.

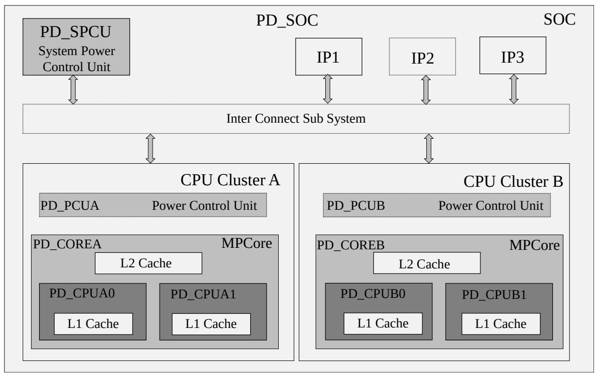

Figure 1. Example of a Complex SoC (Courtesy: ARM SOC)

The SoC is constructed with 4 CPU instances with 2 CPU cores within a CPU cluster. The default top power domain is PD_SOC. At the SoC level there are 2 clusters A and B and a "System Power Unit (SPCU)", which is a power domain PD_SPCU. Also there are interconnects and other design IPs. The "CPU Cluster A" consists of PD_CPUA0 and PD_CPUA1 power domains that are architected in such a way so that the CPU's can be implemented standalone.

The CPUs in a cluster shares an L2 cache and each CPU has an L1 cache. These caches are considered as separate power domain PD_L2 and PD_L1. In addition, each CPU cluster is also a power domain, e.g. PD_COREA. And each possesses a secondary "Power Control Unit" confined in PD_PCUA power domain. When the PD_COREA is turned OFF, then PD_CPUA0 and PD_CPUA1 power domains are also turned OFF. The shared L2 cache can be ON when any of the CPU is ON. The cluster level domain PD_COREA has ON, OFF, RET power states and the core level CPUs power domain PD_CPUA0 also has ON, OFF and RET power states. So the hierarchical and power domain boundaries are visibly clear through the color gradation from Figure 1.

Now, let's define the power states of our SoC starting from the cluster CPUs. When using **add_power_state** the recommended approach is first to create power states of the supply sets of the power domain and then to create the power states of the power domain, in terms of its supply sets, as well state of any dependent power domains. This hierarchical composition makes it possible for a SoC architect to relate the power states of dependent IPs (includes third party IP) to the "System Level Power States". Just to note, there is no mechanism in the UPF language to do the same using PSTs which is discussed detail in Section II.

The power states of supply set for the PD_CPUA0 power domain is denoted by PD_CPUA0.***primary,*** and the states are as follows.

Example 1. Power States of a Supply Set
```
add_power_state PD_CPUA0.primary
-state {ON -simstate NORMAL -logic_expr {pwr_ctrl==1} -supply_expr {power=={FULL_ON,1.0} &&
ground=={FULL_ON,0}} }
-state  {OFF  -simstate  CORRUPT  -logic_expr  {pwr_ctrl==0}-supply_expr  {power==OFF  &&
ground=={FULL_ON,0}}}
```

Hence the **add_power_state** for the supply set can be defined through ***-logic_expr***, ***-supply_expr*** and ***-simstate***. The logic expression is a Boolean expression defined in terms of logic ports, logic nets, interval functions, and power states of the given supply set or supply set handle (e.g. `PD_CPUA0.primary`). The supply expression is also a Boolean expression that may reference available supply nets, supply ports, and (or) functions of supply sets or supply set handles. Since the PD_CPUA0 supports retention strategies, hence before defining the power states definition through the power domains, it is required to specify the fundamental states of the retention supply set.

Example 2. Defining Supply Set (Retention) for PD_CPUA0
```
create_power_domain PD_CPUA0
-supply {retention}
…
```

```
add_power_state PD_CPUA0.retention
- state {ON -simstate NORMAL  -logic_expr  {retpwr_ctrl}  -supply_expr {} }
- state {OFF -simstate CORRUPT -logic_expr {!retpwr_ctrl}  -supply_expr{}}
```

Example 3. Power States of a Power Domain PD_CPUA0

```
add_power_state PD_CPUA0
-state {ON -logic_expr{PD_CPUA0.retention ==ON && nRETCPU0==1 && PD_CPUA0.primary==ON}}
-state {RET -logic_expr{PD_CPUA0.retention ==ON && nRETCPU0==0 &&PD_CPUA0.primary==OFF}}
-state {OFF -logic_expr{PD_CPUA0.retention ==ON && PD_CPUA0.primary==OFF}}
```

In order to complete the power state constructions of the SoC in Figure 1 at cluster level, and utilize the same throughout this paper; let's extend the hierarchical composition of PD_CPUA0 up to it parents PD_COREA. The power state for PD_COREA is shown in Example 4.

Example 4. Power States of a Power Domain PD_COREA

```
add_power_state PD_COREA
-state {ON -logic_expr {primary == ON}}
-state {RET -logic_expr {primary == ON && PD_CPUA0 == RET && PD_CPUA1 == RET}}
-state {OFF -logic_expr {primary == OFF && PD_CPUA0 == SHD && PD_CPUA1 == OFF}}
```

In above two Examples 3 and 4, the **add_power_state** for the power domain is defined through only ***-logic_expr*** which is simply defined in terms of power states of supply sets or supply set handles, and (or) power states of lower level power domains, logic nets etc. but without any -***supply_expr*** or -***simstate***. From the example it is evident that ***logic_expr*** and ***supply_expr*** are design logic and power supply conditions respectively that cause the power state to become (or remain) active.

It is also distinctive that the -***simstate*** option specifies the power related behavior of the logic expression of the power domain and implicitly connected to the supply set. Hence –***simstate*** plays a vital role in PA verification. When a simstate of a supply set is the primary power of a given power domain, it actually determines the simulation behavior of all the design elements in that domain. The valid values of simstate are listed below.

List 1. Valid State Values of Simstate

```
NORMAL,
CORRUPT,
CORRUPT_ON_CHANGE,
CORRUPT_STATE_ON_CHANGE,
CORRUPT_STATE_ON_ACTIVITY,
CORRUPT_ON_ACTIVITY, and
NOT_NORMAL.
```

These state values bears specific significance for having more sensitivity to simulation changes that could cause corruption. The NOT_NORMAL simstate actually provides the flexibility to allow incremental refinement to any other -***simstate*** other than NORMAL through **add_power_state -*update*** commands and options.

In the Syntax 1, the -***update*** indicates that this option provides additional information for a previously defined power state that has the same <object_name> and is targeted to be executed within the same scope. Hence it is possible to define a set of power states for a power domain or supply set and refine it incrementally though –***update.***

It is clear now that the incremental refinement concepts of power states allows an already defined power state to be refined through its objects if required or on demand. They derive sets of power states for different design abstraction levels starting from RTL and applicable up to the PG-netlist through constraint, configuration and implementation UPF.

B. Background and Motivation

Numerous papers have been published in last two years, 2015-16, focusing on functionality, features, and refinement capabilities of power states for UPF objects [3], [6], [9], [10]. However the foundation of power states are not focused in a ground up manner. The considerable growth in low power design has shifted the PA verification paradigm towards early design phases of DVIF. This in turn pushes the limits of legacy PST to an ultimate end. And UPF 3.0 exactly starts from that end point with a complete and new concept of power states. But concepts remain sketchy until ground breaking works actually start to create the physical structures.

In this paper, we are motivated with the objective to create the foundation of power states from the ground up, with real examples and case studies that are actually incorporated and executed through PA static and PA dynamic verification tools. We further hope, the empirical results from our studies will remain as the first source of physical interpretation and implementation proof of the fundamental power states concepts.

C. Organization of this Paper

This paper is organized in the following structure. Section II shows the PST limitations by using examples and experimental results. The section III establishes the conceptual set of fundamental power states that drives the core of UPF modeling and power aware verification, virtually for any level of design abstraction from RTL. The case studies on different complex SoC environments are explained in section IV. The final section V summarizes the results of the case studies and concludes the paper. The references are shown at the end.

II. LIMITATIONS OF POWER STATE TABLES

D. Definition of a Power State Table (PST)

A power state table (PST) defines the legal combinations of port states, i.e., those combinations of port states that can exist at the same time during operation of the design. As mentioned earlier in Section I-A, PST is typically defined with three different UPF commands, **'add_port_state', 'create_pst', 'add_pst_state'** and their attendant options. Their combination defines the states of physical supply ports and physical supply nets to specify particular ordering of supply nets to denote a specific mode of design operation. PST based analysis of these possible combinations of power supply values enables UPF-supporting tools to perform static analysis so as to determine isolation and level shifting requirements of a given design.

Example 5. Sample PST for only a few power domains

```
# Power States
add_port_state VDD -state {ON 2.24}
add_port_state VDDCOR -state {ON 2.24} - state {RET 1.17} - state {OFF off}
add_port_state VDDA0 -state {ON 2.24} - state {RET 1.17} - state {OFF off}
add_port_state VDDL2 -state {ON 2.24} - state {RET 0.77} - state {OFF off}
add_port_state VDDL1 -state {ON 2.24} - state {RET 0.77} - state {OFF off}
#PST
create_pst soc_pt              -supplies {VDD VDDCOR VDDCA0 VDDL2 VDDL1    VSS}
add_pst_state sd   -pst soc_pt -state    {ON   ON     ON     ON    ON      ON}
add_pst_state sretl2 -pst soc_pt -state  {ON   ON     RET    RET   ON      ON}
add_pst_state sretl1 -pst soc_pt -state  {ON   RET    ON     RET   ON      ON}
add_pst_state stdby -pst soc_pt -state   {ON   ON     RET    RET   RET     ON}
add_pst_state slp -pst soc_pt -state     {ON   ON     ON     ON    ON      ON}
```

When the low-power design has a small number of power domains and operating modes, apparently it is easy and straight forward to represent the power modes and combinations in a PST as shown in above Example 5. However, the PST tends to overflow with state space issues when the power modes or number of power domains increases. Even though this overflow is physical and apparent for readability issues, which can be broken down in to smaller tables. But there are more critical issues involved with the PST with increasing number or depth of hierarchical power domains. It may cause at least the following two issues listed in List 2.

List 2. PST Issues Involved with Increasing Hierarchical Power Domains

- When the source and sink lies in different hierarchies, states of lower domain need to be duplicated at the top level for the tools to allow static analysis.
- Verification tools have to imply brute force algorithm for PST composition when the power domain at the top requires to communicate with a lower sub-domain.

The following example further shows the limitation of physical and implementation issues of PST for a considerable number of power domains.

Table 1. Sample Example of PST for State Space Readability, Physical and Hierarchical Inferring Issues [Composed PST ID: [CP51]]

```
Composed PST Table ( PSTs ):
+--------------+------------------------------------------------------------------------------------------------+
| [CP51]       | eagle_Vpll_pst | eagle_VscuL2_pst    | eagle_Varm_cpu1_pst   | softIP_V_pst            | Vcpu_pst  | Vss_pst
+--------------+------------------------------------------------------------------------------------------------+
| [CP51_CS0]   | pll_on         | scuL2_OPP0          | cpu1_OPP0             | ape_25_pll_on_OPP0      | off_s1    | on_s1
| [CP51_CS1]   | pll_on         | scuL2_OPP1          | cpu1_OPP1             | ape_25_pll_on_OPP1      | on_s1     | on_s2
| [CP51_CS2]   | pll_on         | scuL2_OPP2          | cpu1_OPP2             | ape_25_pll_on_OPP2      | on2_s2    | on_s3
| [CP51_CS3]   | pll_on         | scuL2_OPP3          | cpu1_OPP3             | ape_25_pll_on_OPP3      | on1_s2    | on_s4
| [CP51_CS4]   | pll_on         | scuL2_OPP_PLM0      | cpu1_OPP_PLM0         | ape_50_pll_on_OPP_PLM0  | on3_s1    | on_s5
| [CP51_CS5]   | pll_on         | scuL2_OPP_PLM1      | cpu1_OPP_PLM1         | ape_50_pll_on_OPP_PLM1  | on4_s2    | on_s6
| [CP51_CS6]   | pll_on         | scuL2_OPP_OD0       | cpu1_OPP_OD0          | ape_100_pll_on_OPP_OD0  | on1_s2    | on_s7
| [CP51_CS7]   | pll_on         | scuL2_OPP_OD1       | cpu1_OPP_OD1          | ape_100_pll_on_OPP_OD1  | on_s1     | on_s8
| [CP51_CS8]   | pll_on         | scuL2_OPP0          | scuL2_OPP0_cpu1_off   | ape_25_pll_on_OPP0      | off       | off
```

As shown in the above example, when the number of the constituent parts of the PST composition is high, it can result in state space explosion issues. To note, the state space explosion issues are not only physical but also subtle in inherent constructions that are exposed during verification tool analysis as shown in above Table 1. Although different static verification heuristics may reduce such state space explosions, like composition of power states only between running source-sink communication models on the fly, instead of brute-force mechanism. But these do not resolve other issues, like duplication of states, state dependency graphs between top and sub-domain, refinement of states when required, identifying illegal states, avoid miss interpretation of state transition coverage, etc.

Even worse, there is no possible combination of UPF commands for PST to provide notions of power states at higher levels of design abstraction (unless physical power ports and nets are available), nor an allowance for the refinement of these states incrementally, on-demand (even when physical power ports and nets are available).

Though there is an exception, when '**add_port_state'** is replaced with '**add_power_state**' to represent power-state at higher level in combination with the other two commands **create_pst'** and **'add_pst_state'** to form a PST; however, the ultimate benefit of incremental state refinement is still not achievable in such cases, as shown below.

Example 6. Mixing of **add_power_state** with PST Commands

```
# Power States
add_power_state ss1 -state ON {-supply_expr (power == {FULL_ON, 2.24}) && (ground ==
{FULL_ON,0.0})}
add_power_state ss2 -state RET {-supply_expr (power == {FULL_ON,0.77}) && (ground ==
{FULL_ON,0.0})}
add_power_state ss3 -state OFF {-supply_expr {(power == {OFF}) && (ground == {FULL_ON, 0.0})
# PST
create_pst soc_pt -supplies {ss1.power ss1.ground ss2.power ss2.ground ss3.power ss3.ground}
add_pst_state sd   –pst soc_pt –state     {ON      ON     ON     ON     ON     ON}
add_pst_state sretl2 –pst soc_pt –state   {ON      ON     RET    ON     ON     ON}
add_pst_state sretl1 –pst soc_pt –state   {ON      ON     ON     ON     ON     ON}
add_pst_state stdby –pst soc_pt –state    {ON      ON     RET    ON     RET    ON}
add_pst_state slp –pst soc_pt –state      {ON      ON     ON     ON     ON     ON}
```

UPF LRM does not specify this mixing as illegal; however it is distinct that **add_power_state** construction with ***–supply_expr*** invites the rigidity in abstracting complete power models to denote power states in the object space and refine them over time either on the original or on its derivative entities. This is further discussed in Section III. Hence, the motivation behind the replacement of PST became obvious.

E. Known Limitations of Power State Tables (PSTs)

PSTs depend on upfront information of supply ports and supply nets and for a large SoC, this information become available when the process technology and detailed supply network is known. This severely delay the point at which power aware verification can start. In addition, it is usually necessary to combine PSTs of sub-domains at the SoC level but that can lead to state space explosion. There are other known issues that compelled PST to become legacy. The following is a comprehensive list of PST limitations.

List 3. Limitations of PSTs

```
-   PST states are defined based on supply net state only
-   Logic net states cannot be used
-   No logical view of the system states: Power domain states cannot be specified
-   Complex PST composition is not possible
-   No support for hierarchical reference to PST state
-   No support for refinement of states
-   Reduction of states is not possible once defined
-   Poor readability
```

III. Concepts of Fundamental Power States

In general, UPF 3.0 LRM specifies any power state as "named power state" that is defined using **add_power_state** for a supply set, power domains, composite domains, groups, models, and instances as shown in the syntax listed in Syntax 1. In addition, a named power state may also be defined for following UPF attributes, originating from -***simstate*** state values.

List 4. Additional Named Power States

```
-   Predefined power state with ON and OFF states for supply sets, and
```

- Predefined power states with UNDEFINED and ERROR states for all objects that have power states.

To note, for a supply set, the power states ON and OFF are predefined as deferred power states. Power state ON is defined with simstate NORMAL; power state OFF is defined with simstate CORRUPT. The definitions of these states may be updated to specify a logic expression, a supply expression, or legality of the state and are further explained in succeeding sections. Regarding ***-simstate*** state values, to note that there are two additional predefined or default power states values, UNDEFINED and ERROR. These are defined for every supply set or all UPF objects. When a supply set is defined without any ***-simstate***, the status of supply set turns to UNDEFINED. However the simstate of ERROR state is actually denoted as a CORRUPT state.

F. Fundamental Power States

The fundamental power state is the rudimentary part of a power state that is not a derivative or product of refinement of any other power state for an object. Fundamental power states of a given object are mutually exclusive and do not refer to any other power state for the same object. The following example explains the roots of fundamental power states of an object (i.e. power domain PD_CPUA0) in light of UPF allowed attributes for defining named power states.

Example 7. UPF Snippet Example of Fundamental Power State

```
add_power_state PD_CPUA0 -domain \
-state {UNDEFINED -logic_expr {PD_CPUA0 != RUN && PD_CPUA0 != SHD} } \
-state {RUN -logic_expr {primary == ON} } \
-state {SHD -logic_expr {primary == OFF} } \
-state {ERROR -logic_expr {PD_CPUA0 == RUN && PD_CPUA0 == SHD} }
```

So fundamental power states are named power states. The example explains UNDEFINED, ERROR and two other RUN and SHD power states for the PD_CPUA0 domain that are not refined at this point. Predefined power states "UNDEFINED" and "ERROR" represent situations in which the set of power state definitions for an object is either inconsistent or incomplete. UPF 3.0 LRM restricts the fundamental power states active status through the following guidelines.

List 5. UPF Guidelines for Specifying Fundamental Power States

- A fundamental power state of an object will remain active whenever any refinement of that power state is active.
- Two different refinements of the same power state must be mutually exclusive.
- The predefined ERROR power state represents the error condition in which two states that should be mutually exclusive are both active at the same time.
- The current power state of an object will be considered as ERROR state, when two different fundamental states of the same object or two different refinements of the same power state are active at the same time,
- The current power state of an object will be considered as UNDEFINED state, when the set of power states for that object is specified as complete.
- Two fundamental power states of the same object must not remain active at the same time.
- Defining a new fundamental power state is not allowed after the power states are marked complete.

The refinement of fundamental power states through ***–update*** depends on several factors including their mutual-exclusion, overlapping or non-mutual exclusion, conjunction, and disjunction of refinement possibilities and status.

G. Conceptual Set of Power States

The fundamental power state refinement concept extends the UPF specification and associated PA verification boundary to early stages of the RTL design and virtually allows verification at any level of design abstraction. The concept allows an already defined power state to be refined through the objects on demand and derives sets of power states for different design levels of abstraction from RTL to PG-netlist. The refinement concepts are actually derived from the fundamental power states definition and their implication guidelines. The conceptual sets [1], [3] of refining power states are listed in the List 6.

List 6. Conceptual Sets of Fundamental Power States

1. Definite Power States,
2. Deferred Power States, and
3. Indefinite Power States.

Let's explain the UPF LRM definitions of these fundamental power states through examples.

Definite Power States: A definite power state is a power state whose defining expression (i.e. –***logic_expr***{}) is made up of a single term or conjunction of terms (i.e. usage of only "&&" operator) such that each term is one of the following:

- A Boolean expression over signal in the design or
- A term of the form <object> == <state>, where <object> is the name of an object for which power states are defined and <state> is the name of a definite power state of <object>.

Example 8. UPF Snippet Example of Definite Power States

```
# Definite power state for power domain
add_power_state PD_COREA -domain \
-state {RUN -logic_expr {primary == ON}} \
-state {SHD -logic_expr {primary == OFF}}
# Definite Power State with Design Signals or Controls in Logic Expression
add_power_state PD_COREA \
-state ON_HIGH {-logic_expr {(PD_COREA.primary == RUN && PD_CPUA0.primary == INT_ON &&
PD_L2.primary == RAM_ON && MCTL_ON == 1'b0)}}
```

From the above Example 8, it is possible to conclude that the definite power state defining expression consists of a single or conjunction of terms that may be constructed with <object>==<state> or it may be a Boolean expression over signals and other power domain status in the design. It is important to note that when -***simstate*** is not specified, the verification tool treats the expression as NORMAL. However, for UPF strategies, i.e. isolation analysis, either implementation UPF or ***–simstate*** is required, which is further discussed in Section IV.

Deferred Power States: A named power state is deferred if it has no defining expression (i.e. no ***–logic_expr***{}). The "deferred" states defining expression depends on precise design implementation parameters and remains deferred until later stages of design abstraction. However, eventually deferred states are updated later through the process termed *refinement in place* (actually through -***update*** in **add_power_state** command) and resolved into definite states. The following example shows the initial state or status of deferred power states definition.

Example 9. UPF Snippet Example of Deferred Power States

```
add_power_state PD_COREA.primary -supply \
-state {RUN -simstate NORMAL} \
-state {SHD -simstate CORRUPT}
```

The deferred power state is just a container of -***state*** names and possible speculation of simulation behavior with –***simstate*** options. However these deferred power states are more relevant for **power_expr <** power_expression>; as shown in syntax 1, which specifies the power consumption of an object for the power state, or a function for computing the power consumption. UPF 3.0 LRM specifies that a power expression must be specified only for a deferred power state. Since power consumption is out of the scope of this research, let us provide example of deferred power states more relevant to refinement.

Example 10. UPF Snippet Example of Deferred Power States Relevant to Refinement

```
add_power_state PD_COREA.primary \
-state ON {-logic_expr {ln3 == 1} -simstate NORMAL} \
-state SHD {-logic_expr {ln3 == 0} -simstate CORRUPT}
```

The refinement relevant deferred power state in above Example 10 contains the design control signal and simstate to denote more complete reference of power states for PD_COREA, rather than a container of state. The voltage values, exact names of power supply nets, any other physical or technology dependent parameters, like power switches etc. for defining the Boolean expression for the –***logic_expr*** and (or) -***supply_expr*** are usually become available only during implementation (synthesis or post synthesis level) of the design. Hence for deferred power states, –***simstate*** is equally important along with HDL signal expression, specifically for UPF strategy (i.e. isolation) analysis until the implementation UPF became available.

Indefinite Power States: An Indefinite power state is a named power state that is neither a definite power state nor a deferred power state.

Example 11. UPF Snippet Example of Indefinite Power States

```
add_power_state PD_COREA -domain \
-state {UNDEFINED -logic_expr {PD_COREA != RUN && PD_COREA != SHD} }
```

Obviously, as observed in the indefinite power state in Example 11 above, the UNDEFINED state value specifying the power state for the Boolean expression ***-logic_expr*** {PD_COREA != RUN && PD_COREA != SHD} }will never satisfy to refine. As well it is also clear that indefinite power states are usually constructed with disjunction of expressions like, <object>!=<state>.

Thus it is also clear that the definite power states corresponds to state refinements for higher levels and deferred power states for lower level of design abstraction. This is because at a higher level, initially objects remain undefined by default unless defined with definitive power states. However once defined, it allows the definite power states to refine by derivations or be defined through a complete new states or sub state, based on its original definition (actually through updating the expression in ***–logic_expr***).

The following three Examples 12.1, 12.2 and 13 explains the details of *refinement by derivation* and *refinement in place* which are applicable to define and model the "definite" and "deferred" power states respectively.

Example 12.1 Refinement by Derivatives: Applicable for Definite Power States

```
# Definite power state Refinement for power domain
add_power_state PD_COREA  -domain –update \
-state NEW_RUN {-logic_expr \
       {(power == {FULL_ON, 1.1} ) && (ground == {FULL_ON, 0.0})}}
```

From Example 8, for the definite power states, the previous ***-state*** {RUN ***–logic_expr*** {***primary*** == ON}} now updated with the ***–state*** NEW_RUN {(…) && (…)} as shown in the Example 12.1 above. This approach involves defining a new power state, with a new ***–state*** name and updated ***-logic_expr***, based on the original power state. The refinement by derivation for the other example (i.e. # Definite Power State with Design Signals or Controls in Logic Expression) given for definite power states in Example 8, based on ***–logic-expr*** with a logic or control signals is also given below.

Example 12.2 Variant of Refinement by Derivatives: Applicable for Definite Power States

```
# Definite Power State Refinement with Design Signals or Controls in Logic Expression
add_power_state PD_COREA.primary –supply –update \
-state ON.ON_STATE {-logic_expr {(PD_COREA.primary == RUN && MCTL_ON ==1'b1)}}
```

The refinement by derivation preserves the original power state definition and thus avoids unexpected semantic changes in other commands that refer to that original power state. This kind of refinement results in an refinement of hierarchy in which more abstract states are refined to create more specific states. Hence the refinement by derivatives reinforce the capabilities of creation of overlapping or non-mutually exclusive, new and independent power states from a more abstract state.

Example 13. Refinement in Place: Applicable for Deferred Power States

```
add_power_state PD_COREA.primary -supply  -update\
-state {RUN -logic_expr {nPWRUP_CON==1'b0}}
```

In contrast the *refinement in place* as shown in Example 13, is similar to how ***-update*** works for **add_power_state** or in the case of **create_power_domain** as specified by UPF LRM [1], [2]. But updating the power state through ***–update*** implies that it actually modifies the original definition rather than creating a new definition or a new power state. So it is obvious that deferred power states eventually evolve to definite power states in the course of the design verification and implementation flow.

However, the evolution of deferred to definite power states is only true in perspective of the inherent meaning of the definite power state. The physical entity of a deferred power state even after refinement remains different, because of two distinctive reasons. The first is, deferred to definite refinement does not create any new state for supply sets or power domains or other objects, and the second is, the boundary of mutual-exclusion conditions and transition between original and refined states are indistinct.

Actually such refinements impose additional pressure on PA static verification to perform Boolean analysis of expressions. In fact, the power state refinement methodology encourages the definition of mutually exclusive power states. The deployment of static or (and) dynamic verification tools can only ensure whether refinement of definite-to-definite and deferred-to-definite power states provides expected results [4], [5]. And it is also distinctive that the indefinite power states do not fall in the deferred or definite categories. Therefore indefinite states are never refinable and it is usually recommended to avoid them in UPF modelling for any design.

H. Why Indefinite Power States Are a Misnomer?

If indefinite power states are not required or recommended, then the obvious question is why UPF LRM defines such a state [2]. The ultimate answer may converge to the fact that it is defined to avoid it. Primarily, indefinite states arise from state and object inequality (!=), negation (!), exclusion (||) operator in ***-logic_epr***. Secondarily it may refer to an indefinite state of a dependent or another object. Ultimately they do not provide the option of refinement. The artifacts can be explained through the following examples. Let us consider that PD_CPUA0 initially defines RUN as an indefinite state as shown below.

Example 14. Artifacts of Indefinite Power States

```
add_power_state -domain PD_CPUA0 \
-state {RUN -logic_expr {primary != OFF }}
```

The subtle meaning may refer to the fact that the state RUN may become active when primary is not an OFF state, but probably in ON state or BIAS state etc. But such an expression is fine if they are not intended to refine later.

Typically, indefinite states are used when composing top level power domain power states based on lower sub domain level power states due to its convenience, as shown in Example 15 below.

Example 15. Artefacts of Indefinite Power States

```
# Sub domain PD_CPUA0 power state
add_power_state -domain  PD_CPUA0
    -state {RUN -logic_expr {…}  \
    -state {RET logic_expr {…}  \
    -state {OFF logic_expr {… }
# Sub domain PD_L2 power state
add_power_state –domain PD_L2
    -state {RUN logic_expr {…}   \
    -state {RET logic_expr {… }   \
    -state {OFF logic_expr {…}
# Top domain PD_COREA power state
add_power_state -domain PD_COREA
    -state {ON -logic_expr{  { PD_L2  !=OFF }  ||  {PD_CPUA0 == RUN }}
```

The ON state for PD_COREA is active when PD_L2 is either in RET state or in RUN state or when PD_CPUA0 is in RUN state. It is clear that there are many overlapping possible states of PD_L2 and PD_CPUA0 for the ON state of PD_COREA. So, this is an indefinite power state because of operators (!=, ||) used in the ***–logic-expr***. Despite that it is facilitating the coordination of state for hierarchical dependency, the user needs to be aware that it implies overlapping or non-mutually exclusive states. In most cases, it is possible to rephrase the expression of ON state to be in the definite power state status for PD_COREA. Alternatively it could be intended that all indefinite states are don't care while definite states are state to care. For such cases, it may also be useful to mark the indefinite power states with UNDEFINED state value defined by UPF 3.0 LRM [2] as shown in Example 11 for indefinite power states.

I. The Flexibility and Controllability of Definite and Deferred Power States

In the previous sections we observed the intrinsic flexibility of UPF through specifying power states for power domains and supply sets through the ***-logic_expr, -supply_expr*** and ***-simstate*** etc. It is also observed that the power state for power domain and supply set allows users to implicate any possible combination of ordered list of power states through the definite and deferred states definition. It is further required to explore the flexibility and controllability of power states in order to comprehend efficient UPF modeling. The flexibility and controllability of the definite and deferred power states other than the gifted *incremental refinement* feature, comes in a different format and flavour. For example, the hierarchical composition of power states of power domains can be specified in terms of power states of its supply sets.

It is also required to fully comprehend the power states defining mechanism for hierarchical power domains, specifically to grasp the complex communication and dependency in defining power states for such power domains.

Recalling Figure 1, Examples 8 for definite, and Examples 10 for deferred power state, let us reconstruct the power states of the power domains at top level of the SoC design sub-system, expressed in terms of both of its supply set handles and power states of the dependent power domains. This is shown below in Example 16 and Example 17.

Example 16. Power States for Hierarchical Domains with Domain and Supply Set Handle

```
# Sub domain PD_CPUA0 power states through <object_name> power domain
# Definite power state for power domain
```

```
add_power_state PD_CPUA0 -domain \
-state {RUN -logic_expr {primary == ON} } \
-state {SHD -logic_expr {primary == OFF} }
# Sub domain PD_CPUA0 power states through supply set handle
# Deferred power states through supply set handle
add_power_state PD_CPUA0.primary -supply \
-state {RUN -simstate NORMAL} \
-state {SHD -simstate CORRUPT}
# Optional but used to update a new state in sub domain PD_CPUA0
add_power_state PD_CPUA0 -domain -update \
  -state {RET}
```

The above Example 16, explains the power state defining and update mechanism of a sub domain PD_CPUA0, through –***domain*** as <object_name>. A variation of the definition is also shown with the supply set handle PD_CPUA0.***primary*** through –***supply*** options.

Example 17. Power States for Hierarchical Domains with States of the Dependent Power Domains

```
# Sub domain PD_CPUA0 states dependency on PD_COREA power states
add_power_state PD_COREA -domain \
  -state {RUN -logic_expr {primary==ON  && PD_CPUA0==RUN}} \
  -state {SHD -logic_expr {primary==OFF && PD_CPUA0==SHD}} \
  -state {RET -logic_expr {primary==OFF && PD_CPUA0==RET}}
# Top domain PD_SOC states dependency on Sub Domain PD_COREA power states
add_power_state PD_MPCore -domain \
  -state {RUN -logic_expr {primary==ON  && PD_L2==RUN && PD_COREA==RUN}} \
  -state {DMT -logic_expr {primary==OFF && PD_L2==RUN && PD_COREA==SHD}} \
  -state {SHD -logic_expr {primary==OFF && PD_L2==SHD && PD_COREA==SHD}}
```

Example 17 shows the hieratically higher power domain PD_COREA power state's dependency on a lower sub domain PD_CPUA0 states {... PD_CPUA0==RUN} as well the top power domain PD_SOC state's dependency {… PD_COREA==RUN} on sub domain PD_COREA power states. This is one type of flexibility.

In an IP centric design flow, **add_power_state** provides a mechanism to easily leverage the power states of IP blocks or lower level blocks when defining the states of a power domain at a higher level. We already discussed power state refinement in previous Examples 12.1, 12.2 and 13. The IP provider may describe a few fundamental power states for their IP; however the SoC integrator may choose to further refine some of these states to better fit in with the SoC power modes or states in the hierarchical power state definition. This is also another form of flexibility.

The UPF 3.0 LRM also introduces a grouping mechanism which allows the user to do similar composition at a convenient design hierarchy level for very large and complex SoCs. The UPF **create_power_state_group** command defines a group name that can be used in conjunction with the **add_power_state** command. A power state group is used to collect related power states defined by **add_power_state**. The -***group*** (<group_name>) is defined in the current scope. The power state group defines the legal combinations of power states of other objects in this scope or in the descendant subtree. That is, it represents the combinations of states of those objects that can be active at the same time during operation of the design.

Example 18. Grouping of legal Power States

```
create_power_state_group PG_SOC
add_power_state -group PG_SOC \
 -state {RUN -logic_expr {primary==ON  && PD_L2==RUN && PD_COREA==RUN}} \
 -state {DMT -logic_expr {primary==OFF && PD_L2==RUN && PD_COREA==SHD}} \
 -state {SHD -logic_expr {primary==OFF && PD_L2==SHD && PD_COREA==SHD}}
 -state {RET -logic_expr {primary==OFF && LP_RET1N == 1'b1 PD_COREA == OFF}} \
 -state {OFF -logic_expr {primary==OFF && LP_RET1N == 1'b0 PD_COREA == OFF}}
add_power_state -group PG_SOC -update \
 -state "RET.PD_CPUA0_ACT -logic_expr {PD_COREA.PD_CPUA0.ACT} -illegal" \
 -state "OFF.PD_CPUA0_ACT -logic_expr {PD_COREA.PD_CPUA0.ACT} -illegal"
```

The power state also provides a mechanism to control the state space of power modes of design operation through explicitly or implicitly marking state legality. The legality of power states can be specified through –***illegal*** options during –***update*** or through -***complete*** which signifies all undefined power states beyond, are illegal. This is explained in the following examples.

Example 19. Legality of Power States
```
#Explicitly specifying illegal power states
add_power_state PD_COREA-update \
-state {PD_CPUA0_RET_ONLY -illegal   \
-logic_expr {primary == ON && PD_L2 == RUN && PD_CPUA0 == RET}}
#Implicitly specifying of illegal power states
add_power_state PD_COREA –update –complete
```

The above mentioned power states for hierarchical domains with states of the dependent power domains, grouping of related power states, and legality or completeness of state definition also facilitates other aspects discussed in succeeding examples. One of the most obvious feature of deferred power states is incremental refinement – which may refer to the intrinsic flexibility of UPF to associate a power domain with supply set and implicate any combination of power states in the successive refinement flow. The constraint UPF requires to define the power states of supply set and power domains.

Example 20. Example of Constraint UPF
```
add_power_state PD_CPUA0.primary
-state {ON -simstate NORMAL}
-state {OFF -simstate CORRUPT}
add_power_state PD_CPUA0.aon
-state {ON -simstate NORMAL}
add_power_state PD_CPUA0
-state {ON -logic_expr {PD_CPUA0.aon==ON && PD_CPUA0.primary==ON}}
-state {RET -logic_expr {PD_CPUA0.aon==ON && PD_CPUA0.primary==OFF}}
-state {OFF -logic_expr {PD_CPUA0.aon==ON && PD_CPUA0.primary==OFF}}
```

The configuration UPF requires to define the logic expression for supply set and power domain.

Example 21. Example of Configuration UPF
```
add_power_state PD_CPUA0.primary -update
-state {ON -logic_expr {pwr_ctrl==1}}
-state {OFF -logic_expr {pwr_ctrl==0}}
add_power_state PD_CPUA0 -update
-state {ON -logic_expr {PDL1 == ON}}
-state {RET -logic_expr {PDL1 == RET}}
-state {OFF -logic_expr {PDL1 == OFF}}
add_power_state PD_CPUA0 -update
-state {ON -logic_expr {ret_ctrl==0}}
-state {RET -logic_expr {ret_ctrl==1}}
-state {OFF -logic_expr {ret_ctrl==0}}
```

Finally the implementation UPF requires that the logic expression for supply set be updated with technology specific information, as shown in Example 22 below.

Example 22. Example of Implementation UPF
```
add_power_state PD_CPUA0.primary -update
-state {ON -supply_expr {power=={FULL_ON,0.6} && ground=={FULL_ON,0}}}
-state {OFF -supply_expr {power == OFF && ground=={FULL_ON,0}}}
add_power_state PD_CPUA0.aon -update
-state {ON -supply_expr {power=={FULL_ON,0.6} && ground=={FULL_ON,0}}}
```

The inherent advantages of defining fundamental power states as discussed so far, may be best summarized as follows.

List 7. Advantages from Power State
- Allows to model UPF, i.e. power management architecture from very early stage of design.
- Allows to integrate design IP any time in the power management architecture.
- Allows to analyze and validate UPF strategy requirements.
- Allows to compute accurate state transition coverage information through interdependent states and
- Prevent intermediate state transitions of definite and deferred power states during refinement.

IV. Case Studies of Fundamental Power States

J. UPF Strategies Analysis and Validation through Definite and Deferred Power States

Referring to the discussion on definite and deferred power states based on Example 8, 9, and 10, it is possible to analyze and validate UPF strategies like isolation requirements through the conceptual sets of fundamental power states. Let us consider the following examples of definite and deferred power states defined in the very early abstraction of RTL when refinement of states or implementation UPF is still not available.

Example 23. Snippet for UPF Strategy Analysis

```
add_power_state PD_CPUA0.primary \
-state ON {-logic_expr {ln3 == 0} -simstate NORMAL} \
-state OFF {-logic_expr {ln3 == 1} -simstate CORRUPT}
add_power_state PD_COREA.primary \
-state ON {-logic_expr {ln3 == 1} -simstate NORMAL} \
-state OFF {-logic_expr {ln3 == 0} -simstate CORRUPT}
add_power_state PD_CPUA0 \
-state RUN {-logic_expr {primary == ON}} \
-state SHD {-logic_expr {primary == OFF}}
add_power_state PD_COREA \
-state RUN {-logic_expr {primary == ON}} \
-state SHD {-logic_expr {primary == OFF}}
```

The static analysis and validation results are represented in Table 2.

Table 2. Static Tool Analysis and Validation for UPF Strategies (e.g. Isolation) Requirements

```
---------------------------------------------------------------------------
1. Power Domain Boundary Analysis [BOUNDARY_ANALYSIS]:
  1. ~/PD_CPUA0 -> ~/PD_COREA [PD1_to_PD2]:
    Isolation: Required
    Level shifter: Not Analyzed
    Details:
      Analysis between supplies:
       Source Supply: ~/PD_CPUA0 Supplies(PD_CPUA0.primary.power,PD_CPUA0.primary.ground)
       Sink   Supply: ~PD_COREA Supplies (PD_COREA.primary.power,PD_COREA.primary.ground)
     PST used for analysis: Composed PST ID: [CP2]
      +-------------+-------------------------------------------+
      | State(s)    | PD_CPUA0.primary[O2] | PD_COREA.primary[O1] |
      |             | {Source}             | {Sink}               |
      +-------------+-------------------------------------------+
      | {[CP2_CS1]} | ON[O2_V0]            | OFF[O1_V1]           |
      | {[CP2_CS0]} | OFF[O2_V1]           | ON[O1_V0]            |Iso
      +-------------+-------------------------------------------+
1. Source power domain: ~/PD_CPUA0 to Sink power domain: ~/PD_COREA.
     Total 3 Missing isolation cells [Total Crossings: 3, Shared Crossings: 0]
     1.1. Source port: ~/q_A [LowConn] to Sink port: ~/q_A [HighConn], width:1
             Total 1 Missing isolation cells [Total Crossings: 1, Shared Crossings: 0]
             1.1.1. Inferred type: ISO_MISSING, count: 1
    Possible reason: 'Isolation is required from (~/PD_CPUA0)=>(~/PD_COREA) and neither isolation
strategy nor isolation cell is present in design' Analysis link: [PD1_to_PD2].
```

It is important to note that the isolation analysis and validation is possible without **set_isolation** strategy definition and implementation UPF at a very early stage of design abstraction.

K. State and Transition Coverage through Definite and Deferred Power States

During the refinement process of fundamental power states, the ***–illegal*** or (and) ***–complete*** options provides useful information for dynamic simulation tool to accurately compute the coverage from states and their transitions. The entire PA sources for coverage metric computation model can be based on the following broad categories, listed in List 8.

List 8. The Complete Source of PA Coverage Metric Computation

```
(1). Coverage information from PA Dynamic Checks based on;
     - PA testbench,
     - Automated PA Sequence Checkers and
     - Custom PA Checkers.
(2). Coverage information from Power states and power state transitions based on;
     - Design controls
     - Supply ports and nets created in the UPF and design
     - Power domains and their power states
     - Supply sets and their states
     - Power Switch State and their Transitions
     - State transitions for ISO, RFF, PSW Control and Acknowledgement signals
(3). Coverage Information from Cross-Coverage based on;
     - All possible combination of interdependent power states,
     - As well their possible simultaneous transitions.
```

However, from the definition of "definite" and "deferred" power state, the potential coverage metric are relevant to the "-Power domains and their power states" and "-Supply sets and their power states" from (2) as shown in the List 8 above. Nevertheless, there is no specific UPF semantic that may allow to device state and transition coverage computation model instantaneously. UPF 2.1 defines the **describe_state_transition** command for monitoring the legality of power state transition from one named power state to another. UPF 3.0 also provide similar semantics through **add_state_transition** [1], [2].

Let us consider the **describe_state_transition** in conjunction to **add_power_state** to demonstrate the coverage collection and computation models for definite and deferred power states.

Example 24. UPF Snippet for Coverage Computation for Power Domains, Power States and State Transitions

```
add_power_state PD_CPUA0.primary \
-state PD_CPUA0_low_volt {-supply_expr {(power == {FULL_ON,3.0}) && (ground ==
{FULL_ON,0.0}) } -simstate NORMAL }
-state PD_CPUA0_moderate_volt {-supply_expr {(power == {FULL_ON, 5.0}) && (ground ==
{FULL_ON,0.0}) } -simstate NORMAL }
-state PD_CPUA0_high_volt {-supply_expr {(power == {FULL_ON, 7.0}) && (ground ==
{FULL_ON, 0.0})} -simstate NORMAL }
-state PD_CPUA0_primary_off {-supply_expr {(power == {OFF}) && (ground == {FULL_ON,
0.0})} -logic_expr {!CPU_PWR_low && !CPU_PWR_high && !CPU_PWR_moderate } -simstate
CORRUPT }
add_power_state PD_CPUA0 \
-state PD_CPUA0_on {-logic_expr {PD_CPUA0.primary == PD_CPUA0_high_volt && PD_L1 ==
PD_L1_on}}
-state PD_CPUA0_ret {-logic_expr {PD_CPUA0.primary == PD_CPUA0_primary_off && PD_L1 ==
PD_L1_ret}}
-state PD_CPUA0_off {-logic_expr {PD_CPUA0.primary == PD_CPUA0_primary_off && PD_L1 ==
PD_L1_off}}
add_power_state PD_COREA \
-state RUN {-logic_expr {PD_CPUA0 == PD_CPUA0_on && PD_L2 == PD_L2_on}}
-state SLEEP {-logic_expr {PD_CPUA0 != PD_CPUA0_on && PD_L2 != PD_L2_on}}
### configure state transition legality ###
describe_state_transition st8 -object PD_CPUA0.primary -from {} -to {}
describe_state_transition st14 -object PD_CPUA0 -from {} -to {}
describe_state_transition st15 -object PD_COREA -from {} -to {}
describe_state_transition st16 -object PD_COREA.primary -from {PD_COREA_primary_on} -to
{PD_COREA_primary_off} -illegal
```

The UPF **describe_state_transition** command option ***-from*** and ***-to*** specify many-to-many transitions. The two arguments accepts unordered list of power state names active before and after a state transition respectively. The -***illegal*** (or –***legal*** by default) is used to tag a state transition legality. The following Table 3 shows the snippet of coverage results obtained through the UPF snippet shown in above Example 24.

Table 3. Snippet of Coverage Computation Results for Power Domains and their Power States

```
---------------------------------------------------------------------------------------
    UPF OBJECT                                            Metric      Goal    Status
---------------------------------------------------------------------------------------
    TYPE: SUPPLY SET /cpu_tester/dut/PD_CPUA0.primary
    State Coverage
        Power State PD_CPUA0_low_volt                    100.0%        100    Covered
            bin ACTIVE                                        3          1    Covered
        Power State PD_CPUA0_moderate_volt               100.0%        100    Covered
            bin ACTIVE                                        1          1    Covered
        Power State PD_CPUA0_high_volt                   100.0%        100    Covered
            bin ACTIVE                                        2          1    Covered
        Power State PD_CPUA0_primary_off                 100.0%        100    Covered
            bin ACTIVE                                         2         1    Covered
        Power State DEFAULT_NORMAL                          100.0%      100    Covered
            bin ACTIVE                                         2         1    Covered
        Power State DEFAULT_CORRUPT                         100.0%      100    Covered
            bin ACTIVE                                         1         1    Covered
    Transitions Coverage                                     50.0%      100       Uncovered
            bin PD_CPUA0_low_volt -> PD_CPUA0_moderate_volt    1         1    Covered
            bin PD_CPUA0_low_volt -> PD_CPUA0_high_volt
    Described Transitions Coverage                           100.0%      100   Covered
            bin st8                                            7         1    Covered
    TYPE: POWER DOMAIN /cpu_tester/dut/PD_CPUA0             100.0%      100    Covered
        Power State PD_CPUA0_on                             100.0%      100    Covered
            bin ACTIVE                                         2         1    Covered
        Power State PD_CPUA0_ret                            100.0%      100    Covered
            bin ACTIVE                                         2         1    Covered
        Power State PD_CPUA0_off                            100.0%      100    Covered
            bin ACTIVE                                         3         1    Covered
    Power State Transitions                                 0.0%        100    ZERO
            bin PD_CPUA0_on -> PD_CPUA0_ret                    0         1    ZERO
            bin PD_CPUA0_on -> PD_CPUA0_off                    0         1    ZERO
    Described Transitions                                    100.0%      100   Covered
            bin st14                                           4         1    Covered
```

Hence the power state coverage metric provides coverage results for a power domain and its associated supply set for both, in terms of “state”, “transition”, and “described transition” coverage, as shown in the snippet results in Table 3. However, the interdependency, depth tree and simultaneous occurrence of fundamental or refined power states are not covered through the above mentioned process

L. Cross-Coverage through Definite and Deferred Power States

The power states and their transitions are usually interdependent in nature, specifically while **add_power_state** is constructed in a hierarchical structure combined with power domain, supply set handles, and states with dependent power domains. As listed in no. (3), in List 7, the coverage metric requires finding their possible cross combination for state-transition occurrence; and hence such coverage is termed *cross-coverage*. Unfortunately UPF LRM does not provide any semantics for computing cross-coverage of power states and their transitions.

The cross-coverage computing model can be best represented by a simple dependency graph. The nodes of the graph will be based on power state logic and supply expressions; where the nodes will represent power states and their edges will represent transitions between the nodes. A path between the nodes will denote a sequence of nodes and edges, connecting a node with its descendant and (or) dependent. This is shown in Figure 2 for a dependent power domain groups.

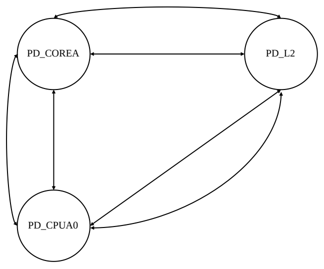

Figure 2. Dependency Graph for Cross-Coverage Computation Model for Power Domains Group PD_COREA->PD_CPUA0->PD_L2 etc.

The power aware simulator requires monitoring and capturing the transitions from the edges of this graph. The path will provide the depth for a group of interdependent nodes and helps to untie the dependency among these nodes. Since there is no UPF semantics, hence we have established the mechanism within the simulation tool to process the cross-coverage metric in conjunction with the following extended methodologies.

Syntax 2. Methodology Extension for Cross-Coverage

```
describe_state_cross_coverage
  [-domains domains_list]
  [-depth cross_coverage_depth]
```

The **describe_state_cross_coverage** command is an augmentation of current UPF intent to supplement the missing *cross-coverage* computation semantic model [6], [7], [11]. The ***–domain*** defines the list of power domains for which cross-coverage needs to be computed. The ***-depth*** is the number of power domains that are involved in the cross-coverage computation and are interdependent. By default, the computation starts with depth one, however the tool will figure out the list of dependent power domains of a particular top or adjacent domains. These dependent power domains, together with the specified top domain, will form a domain group. Then the tool computes the cross-coverage results for this particular group of power domains.

Example 25. UPF Snippet for Cross-Coverage Computation

```
add_power_state PD_L1 \
   -state PD_L1_on {-logic_expr {PD_L1.primary == PD_L1_primary_on}}
   -state PD_L1_off {-logic_expr {PD_L1.primary == PD_L1_primary_off}}
   -state PD_L1_ret {-logic_expr {PD_L1.primary == PD_L1_primary_off &&
PD_L1.default_retention == PD_L1_ret_on}}
add_power_state PD_L2 \
   -state PD_L2_on {-logic_expr {PD_L2.primary == PD_L2_primary_on}}
   -state PD_L2_off {-logic_expr {PD_L2.primary == PD_L2_primary_off}}
add_power_state PD_CPUA0 \
   -state PD_CPUA0_on {-logic_expr {PD_CPUA0.primary == PD_CPUA0_high_volt && PD_L1 ==
   PD_L1_on}}
   -state PD_CPUA0_ret {-logic_expr {PD_CPUA0.primary == PD_CPUA0_primary_off && PD_L1 ==
   PD_L1_ret}}
   -state PD_CPUA0_off {-logic_expr {PD_CPUA0.primary == PD_CPUA0_primary_off && PD_L1 ==
   PD_L1_off}}
add_power_state PD_COREA \
   -state RUN {-logic_expr {PD_CPUA0 == PD_CPUA0_on && PD_L2 == PD_L2_on}}
   -state SLEEP {-logic_expr {PD_CPUA0 != PD_CPUA0_on && PD_L2 != PD_L2_on}}
### configure cross coverage ##
describe_state_cross_coverage -domains {PD_COREA} -depth 3
describe_state_cross_coverage -domains {PD_CPUA0} -depth 2
describe_state_cross_coverage -domains {PD_L2}
```

The ***–logic_expr*** along with the UPF extension in above Example 25, and from the dependency graph from Figure 2, PA dynamic simulator will develop the cross-coverage computation model for PD_COREA->PD_CPUA0->PD_L2 as follows.

TABLE 4 Dependent States for Power Domains

Power Domains	PD_COREA	PD_CPUA0	PD_L2
Power-states	RUN	PD_CPUA0_on	PD_L2_on
Power-states	RUN	PD_CPUA0_ret	PD_L2_on
Power-states	SLEEP	PD_CPUA0_off	PD_L2_off

TABLE 5 Cross-Coverage Data for *-depth*=1 (default) for PD_COREA->PD_CPUA0->PD_L2

RUN -> PD_CPUA0_on -> PD_L2_on
RUN -> PD_CPUA0_ret -> PD_L2_on
SLEEP -> PD_CPUA0_off -> PD_L2_off

Here the TABLE 4 and 5 show the dependent states and default depth of cross-coverage for PD_COREA->PD_CPUA0->PD_L2. The actual results of coverage snippet are given in Table 6.

Table 6. Snippet of Cross-Coverage Results

```
-----------------------------------------------------------------------------------------------
    UPF OBJECT                                        Metric       Goal    Status
-----------------------------------------------------------------------------------------------
TYPE: POWER STATE CROSS
/cpu_tester/dut/PD_COREA(ID:PD1),
/cpu_tester/dut/PD_CPUA0(ID:PD2),
/cpu_tester/dut/PD_L2(ID:PD3)                           33.3%        100    Uncovered
POWER STATE CROSS coverage instance
\/cpu_tester/dut/pa_coverageinfo/PD_COREA/PD_COREA_PS_CROSS/PS_CROSS_PD_COREA
                                                        33.3%        100    Uncovered
Power State Cross Coverage                              33.3%        100    Uncovered
        bin \PD1:SLEEP-PD2:PD_CPUA0_off-PD3:PD_L2_off       0          1    ZERO
        bin \PD1:SLEEP-PD2:PD_CPUA0_ret-PD3:PD_L2_off       0          1    ZERO
        bin \PD1:RUN-PD2:PD_CPUA0_on-PD3:PD_L2_on           2          1    Covered
```

The cross-coverage computation result conforms to the Unified Coverage Interoperability (UCIS) Standards and can be merged with all other PA coverage metrics listed in List 8.

V. CONCLUSION

It appears that PST replacement with **add_power_state** fundamental construct is straightforward and simple, but the change will highly impact the power specification methodologies, power aware verification algorithms, tools, techniques and the entire design, verification and implementation flow. It is also distinctive through our methodical discussion, relevant examples, guidelines, and actual empirical results from case studies that the change is mandatory. Nevertheless, of the enormous impacts, evidently it is possible to go forward with definite, indefinite and deferred power states without the PST. Because definite (indefinite as well) and deferred power states provides the intrinsic flexibility, essential controllability and the gifted features of *refinement* options for contemporary power management in every design today. This study convinces us to assertively recommend to define the "power states" for primary supply set, in terms of logic expression of the design objects. And define the "power states" for power domain, in terms of the states of its primary supply set handles, and when applicable, in terms of the states of any dependent power domains.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to their colleagues, Tom Fitzpatrick and Allan Gordon for providing useful insight and guidance in conducting the research and preparing this paper.

REFERENCES

[1] Design Automation Committee of the IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, "IEEE Standard for Design and Verification of Low-Power Integrated Circuits", Revision of IEEE Std 1801-2009, 6 March 2013.

[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 1801-2015, 5 December 2015.

[3] E. Marschner, J. Biggs, "Unleashing the Full Power of UPF Power States", DVCon' 2015.

[4] D. Prasad, M. Bhargava, J. Bansal, C.Seeley, "Debug Challenges in Low-Power Design and Verification", DVCon' 2015.

[5] M. Bhargava, P. Gairola, "Power State to PST Conversion: Simplifying static analysis and debugging of power aware designs", DVCon' 2016.

[6] S, Wei Tu, T. Lin, A. Feng, C.Y. Ping, "UPF Code Coverage and Corresponding Power Domain Hierarchical Tree for Debugging", DVCon' 2015.

[7] V. Vikram, S. Awashesh K, "Cross Coverage of Power States", DVCon' 2016.

[8] Desinghu PS, A. Khan, E. Marschner, G. Chidolue, "Refining Successive Refinement: Improving a Methodology for Incremental Specification of Power Intent", DVCon' Europe, 2015.

[9] M. Bhargava, P. Gairola, "Power State to PST Conversion: Simplifying static analysis and debugging of power aware designs", DVCon 2015.

[10] P. K. Dwivedi, A. Srivastava, V. Vikram S, "Lets disCOVER Power States", DVCon' 2015.

[11] P. Khondkar, P. Yeung, D. Prasad, G. Chidolue, M. Bhargava, "Crafting Power Aware Coverage: Verification Closure with UPF IEEE 1801", Journal of VLSI Design and Verification, pp.6-17, Vol. 1, November 2017.