2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

UPF Power Models: Empowering the power intent specification

Amit Srivastava, Synopsys Inc, Mountain View, U.S.A. (*asrivas@synopsys.com*)

Harsh Chilwal, Synopsys Inc, Mountain View, U.S.A (*harsh@synopsys.com*)

***Abstract*— Today's low power SoCs have become incredibly complex involving a large number of hard/soft IPs with their own sophisticated power management strategies. As a result, IP integrators must deal with a lot of UPF code for the IPs which needs to be integrated in the SoC environment. This causes an explosion of power intent code due to hundreds and even thousands of instances of some IPs. Such a large amount of power intent code poses several problems in terms of IP integration and functional issues related to power management. This results in costly debug cycles and slow simulation time. In this paper, we propose a methodology of expressing the power intent for the IPs based on power models from IEEE 1801 (UPF) in such a way that eases the burden of IP integration and reduces UPF verbosity. The paper demonstrates how power models can be used to address the problems faced by IP integrators.**

Keywords—Low Power; UPF; Power Models; Hard Macros;Soft Macros;Hierarchical UPF

I. INTRODUCTION

Modern low power System on Chip (SoC) designs have evolved to become incredibly large and complex. They instantiate many IPs with their own power management architecture, captured in IEEE 1801-2015 Unified Power Format (UPF) [1] specification, resulting in significant increase in lines of UPF code. With each IP having their own UPF specification it has become a challenge to integrate the power intent of the IPs into the SoC environment.

Due to the massive design size, IP integrators are faced with new kinds of issues related to file management of power intent of IPs. Issues due to the incorrect association of IP UPF files with the corresponding instance in the design are leading to functional bugs related to power management.

The existing methodologies of capturing power intent require splitting power intent into separate files and then using UPF commands to associate them with each instance of the IPs. This results in the amount of UPF code required for the integration to be directly proportional to the instances of the IPs. Since the IP can have a large number of instances in current SoCs, the amount of UPF code needed for integrating the power intent of IP has also grown significantly thereby increasing the probability of errors. More sophisticated users tackle this by relying on Tcl's scripting capabilities, but the lack of proper debugging tools and absence of self-checking mechanism causes problems in identifying issues related to integration.

In this paper, we propose a methodology which aims to address these challenges by using UPF power models and causes a change in thinking from the existing Hierarchical UPF methodology to model-based representation. The methodology uses built-in semantics defined in the IEEE 1801-2015 Unified Power Format (UPF) [1] language to enable automatic integration for all the instances of the IP, significantly reducing the amount of code during integration. Moreover, the methodology uses UPF's self-checking mechanism to avoid common errors during the integration. We demonstrate the new methodology with real examples and highlight the advantages compared to the existing methodologies with results on real designs.

II. UPF FOR SOCS

Before UPF was first introduced in 2007, power intent of a SoC was captured in a custom side file. With the launch of UPF, the side file approach was dismissed in favor of UPF. However, the power intent for early UPF design used to be concise and flat without any `load_upf` commands. As the power intent for SoCs became

complicated, the flat UPF became verbose and it became a maintainability and scalability issue. The `load_upf` command started getting used as UPF was now partitioned per IP or sub-system. This style is referred to Hierarchical UPF.

A. *Hierarchical UPF*

Hierarchical UPF is written by dividing the power intent into separate files. The partitioning happens mainly at the IP boundary to ensure UPF can be processed as a standalone and can be reused. The UPF commands written in a file represents the power intent for the IP. The file is then loaded into the parent context using `load_upf` command with `-scope` option to provide the instance path of the IP. The tool then processes the UPF commands present in the loaded UPF file and applies them to the instance of the IP. The file-based approach enables the UPF for an IP to be reused. The UPF, however, had to be loaded for every instance of the IP followed by making a connection to supplies and logic controls on per-instance basis. If the UPF file had use of Tcl variables, `load_upf` did not guarantee consistent loading of UPF for every IP instance as Tcl variables could be reassigned between interleaved UPF files.

B. *Modular UPF*

Just like VHDL and Verilog where the design intent is captured in Entity/Architecture or Module level, Power model introduces to UPF, a modular way of writing UPF. A UPF can be written for a logical power model that can be mapped to one or more HDL models (Entity/Architecture or Modules). The tools can process this UPF file in one go for all instances of the model. Modular UPF is applied to one or more instances of a model using `apply_power_model` command and does not use a file name. Instead, the power model name is used to map the model to an instance like HDL model instance. The supply and logic connections can be made via `apply_power_model` command.

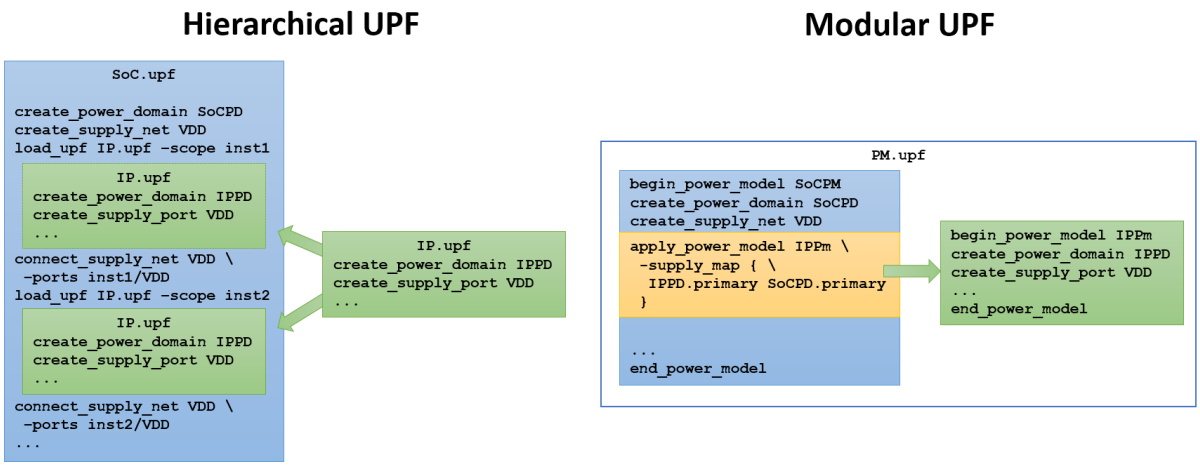

Figure 1: Hierarchical UPF vs Modular UPF comparison

Figure 1 highlights the differences between the Hierarchical UPF and Modular UPF.

III. UPF POWER MODELS

Power models in UPF are model-based representation of the power intent. In that representation, users can describe the power intent for any model or set of models. UPF provides an easy way to encapsulate the UPF commands for an IP and automatically apply to all the instances of the IP using a single UPF command as compared to the hierarchical approach where the user must manually load the power intent for all instances and then connect the supplies individually for each instance of IP.

A. *Creation of power model*

Power models are created by the UPF command `begin_power_model`. Any subsequent UPF commands following this command specifies the power intent of the power model until an `end_power_model` is specified. The power model can be associated with a single model or a set of models using -for option.

In the example below, a power model with the name `upf_model` is created for model `cellA`.

EXAMPLE

```
begin_power_model upf_model -for cellA
create_power_domain PD1 -elements {.} -supply {ssh1} -supply {ssh2}
# other commands …
end_power_model
```

B. *Application of power model*

A power model once defined needs to be applied to the instances of the model present in the design. This is achieved by `apply_power_model` command in UPF. The `apply_power_model` command also allows associating the supplies from the parent context to all the instances of the power model. This is achieved by the `-supply_map` option. If a subset of instances of the IP require different supply connections, then the subset of instances can be specified using `-elements` option.

EXAMPLE

```
apply_power_model upf_model
  -supply_map {{PD.ssh1 ssmain} {PD.ssh2 ssbackup}}
```

C. *Adding parameters to the power model*

Power models can also have parameters defined in them. Parameters are constants which are used within the power model. The parameters can be created by the `add_parameter` command. They can be modified during integration by the `apply_power_model` command with `-parameters` option

Parameters are useful to provide model variations within the families of the IP.

EXAMPLE

```
# Tcl parameters
add_parameter PDNAME -default “PD_PM”
```

IV. KINDS OF POWER MODEL

The power model can be defined for variety of IPs at different abstraction levels. They are Soft Macro, Hard Macro, System-Level IP and Hierarchical model.

A. *Soft Macro*

A soft macro is a soft IP which is represented by the original RTL and UPF. It is typically an IP which is going to be implemented separately in a bottom-up implementation flow. A soft macro is identified by the predefined design attribute `UPF_is_soft_macro` or by UPF command `set_design_attribute` with `-is_soft_macro TRUE`. The example below shows a snippet of the UPF of a soft macro.

EXAMPLE

```
begin_power_model CPU_PowerModel -for {CPU}
 set_design_attributes -model {.} -is_soft_macro TRUE
 create_power_domain PD_CPU -elements {.}
 # ... Other UPF commands
end_power_model
```

B. *Hard Macro*

A hard macro is typically an IP block which is already implemented. For low power IPs, the power management architecture is already present in the hard macro. In such cases, the power intent is required to describe the power management architecture so that the power intent of the external environment can use it. A power model for a hard macro contains a predefined design attribute `UPF_is_hard_macro` or the UPF command `set_design_attribute` with `-is_hard_macro TRUE` option.

For verification purposes, a power model for a hard macro will also have an HDL verification model associated with it. The verification model can be a behavioral model which describes the behavior of the cell without representing internal details. In some cases, the behavioral model does not contain the power management behavior [2]. For such scenarios, the simulation tools can use the information present in the power model to mimic the behavior of the power intent which is not present in the behavioral model.

EXAMPLE

```
begin_power_model memPwrModel -for MEMSRAM_1024X32
 set_design_attributes -models {.} -is_hard_macro TRUE
 # ... Other UPF commands
end_power_model
```

C. *System Level power model*

UPF provides capability to model the power intent for System-Level IP components to be used in System-Level design. These are abstract models that contain power information for calculating power consumption in System-Level simulation. It is identified by the presence of `add_parameter` command with `-type` option.

EXAMPLE [3]

```
begin_power_model dram
 create_power_domain Background -elements {}
 add_parameter vdd -type runtime -default 1.575V
 ...
 add_power_state -domain Background
 -state {precharged_power_down -power_expr {pwr_expr_pre_pdn{vddfreq}}}
 ...
end_power_model
```

D. *Hierarchical Models*

Power models can also be defined for hierarchical models which represent the design and is not marked as a macro. In such case, the power model provides an inbuilt capability of associating the power intent with the HDL model. This can eliminate the reliance on files for splitting the power intent. Moreover, the application of power intent can be achieved by `apply_power_model` which is like VHDL and SystemVerilog where all the external connections, associations and parameter overrides are present in the single command. This is easier to review compared to reviewing adjacent commands in the `load_upf` style of modeling.

EXAMPLE

```
begin_power_model pdmid1 -for {mid}
 create_power_domain PD_MODEL1 -elements {.}
 create_supply_set PM_SS
 apply_power_model pdbot1 \
  -supply_map { \
   {PD_BOT_MODEL1.primary PD_MODEL1.primary} \
   {PM_BOT_SS PM_SS} \
  }
end_power_model
```

V. Challenges With Hierarchical UPF Methodology And How Modular UPF Addresses them

A. *Code Explosion & Tool Performance*

1) Code Explosion

The Hierarchical UPF methodology relies on the mapping between the instance paths and file names. This creates problems when the design has multiple IPs or a large number of instances of one or more IPs. This is a common situation in modern SoCs which integrate the previous generation SoCs as IPs.

SoCs also integrate IPs with UPF from tools like memory compiler as hard macros. The compiler generated UPF files use long and cryptic names that contain some configuration details of the memory.

The problem gets aggravated when different versions of memory are used in the same design. Often there is a small change in the filename which is very hard to notice by human inspection, thereby increasing the chances of incorrect mapping. The instance-based mapping implies there will be a duplication of `load_upf` and corresponding `connect_supply_net` and `connect_logic_net` commands for all the instances of the IP. Any typographical error may result in wrong IP power intent to be associated with the IP instance.

The following example shows an integration of memory hard IP generated by memory compiler tool. Simple alphabets obfuscate the directory pathname in the example. In real designs, the directory structure can be quite deep and consist of large names. In the example, the UPF for memory IP is loaded for the memory instance scope. The supply connections are made after the `load_upf` command.

Example

```
load_upf /a/b/c/d/e/upf/sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00.upf
 -scope U_sms_wrapper1_int_top/U_sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00
connect_supply_net VDD \
 -ports U_sms_wrapper1_int_top/U_sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00/VDD
connect_supply_net VSS \
 -ports U_sms_wrapper1_int_top/U_sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00/VSS
```

2) Overhead from Tcl scripting

The burden of integration can be mitigated to some extent by leveraging Tcl's scripting capabilities. The instances of the IP can be queried using `find_objects` command and assigned to a global variable. The long cryptic filenames can be assigned to simple variable names. The `foreach` loop can be used to iterate over all the instances of the IP and do the `load_upf` followed by subsequent connections. As shown in below example for the previous example.

Example

```
set mem_instances [find_objects \
 -model sadcls0c4l1p32x10m4b1w1c1p1d1r3s2sdz1rw00 \
 -pattern * \
 -object_type inst \
 -transitive TRUE ]
set mem_upf "/a/b/c/d/e/upf/sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00.upf"
foreach instance $mem_instances {
 load_upf $mem_upf
  -scope $instance
 connect_supply_net VDD \
  -ports $instance/VDD
 connect_supply_net VSS \
  -ports $instance/VSS
}
```

Using Tcl scripts can help reduce manual code but it has its own set of problems.

- There are no built-in checks to ensure the correct association is made between IP UPF file and instance.

- The UPF code becomes complex and it is hard to identify issues by human inspection. It requires sophisticated tools that provide advanced debug capabilities to debug the Tcl scripts.
- Use of global variables causes problems when some other script does an unintended modification.
- Often there are scenarios when some instances require some special handling and thus to accommodate it the loop becomes more complex and includes various filters to exclude the instances.
- The queries also introduce performance overhead due to design traversals to discover all the instances of the macro in the design and transfer of large string values of instance paths back and forth between the tool and the Tcl interpreter.

3) Tool performance impact

Apart from performance degradation due to queries, there is severe performance overhead when `load_upf` is used. Since the `load_upf` is designed to be used for each instance, it implies that potentially every instance of the same IP may have their own power intent – separate from other instance of the same IP. This limits the scope for optimizations and gives less flexibility to tools to reuse processing for instances of the IP.

The UPF 2.1 LRM has allowed a list of instances to be specified in a single `load_upf` command. This reduces the chances of variations between IP instances which are specified in the command. However, this doesn't provide full protection and has the same problems as for `load_upf` with a single instance, e.g. a user must manually perform the correlation between IP instances and UPF file.

4) How a power model addresses these problems

With the use of power models, the amount of UPF code can be significantly reduced by using `apply_power_model` command. This allows automatic instantiation of power model for all the instances with a single command invocation. The supply and logic connectivity can be specified in the same `apply_power_model` command. A single `apply_power_model` command is required for IP instances that have same connectivity to the integrating instance. This is illustrated in figure 1. The following example shows how power model can be used to represent the integration of memory IP.

EXAMPLE

```
begin_power_model SRAM_Synch_1Port
 -for {
   sadcls0c4l1p32x10m4b1w1c1p1d1r3s2sdz1rw00
   sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00
  }
 create_power_domain PDP
 create_power_domain PDA_SW -elements {uut} -scope uut
  # ... Other UPF Commands
end_power_model
# This applies for all instances of
# sadcls0c4l1p32x10m4b1w1c1p1d1r3s2sdz1rw00
# sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00
apply_power_model SRAM_Synch_1Port \
    -port_map {{VDD VDD} {VSS VSS}}
```

Since power models provide power intent of a model, which implies that it will be reused for each instance of the model, tools can perform several optimizations including but not limited to single Tcl interpretation of power model and sharing various analysis and semantic checks for all instances of the IP.

The Table 1 summarizes the improvement that VCS compiler [3] demonstrated on two different styles of design where `load_upf` was replaced by power models. We have used SoC RTL designs from two different customers. These designs had a large number of instances of different hard macros and were showing significant overhead in UPF processing during compilation. In these designs, the macro UPFs were converted to power models and `load_upf` commands were replaced by `apply_power_model` commands. We can see that the number of

apply_power_model commands is more than the number of hard macros. This is because there were some groups of instances of hard macro which required different supply connections hence had more number of apply_power_model commands. In each case, we have seen around 2x improvements over the compile time due to various tool optimizations possible with the use of power models.

Table 1: Performance comparison

Customer Scenarios	No. of Hard Macros	No. of load_upf	No of apply_power_model commands	Compilation Time with load_upf	Compilation Time with apply_power_model
Customer #1	338	18051	710	~13 hrs	~6.5 hrs
Customer #2	12	24432	122	~6 hrs	~3 hrs

B. IP Variations

IPs can have minor variations in the power intent imposed by the Integrator IPs. The variations could be related to operating voltages which are captured in the power states definitions. In such case, instead of having different versions of UPF, it becomes easier to parameterize the UPF using variables. The variables are then set with different values before the load_upf of the IP is called. Depending upon the design and methodology, these variables can be set at different locations, often in separate environment files. There is no inbuilt protection from any side-effect and hence it becomes difficult to debug issues related to the setting of incorrect values to the global variables. The UPF 3.0 provides -hide_globals and -parameters option with load_upf command to provide some protection with global variables, but these need to be specified for each instance of the IP which can become tedious and error-prone. Moreover, many EDA tools still don't support these options.

The below example highlights the challenges with parameterization with load_upf. The values need to be explicitly set before load_upf command for SubSystem1.upf and SubSystem2.upf. This is a redundant and error-prone process and increases the amount of code during integration as it needs to be done for all instances of the IP.

EXAMPLE: PARAMETERIZATION USING LOAD_UPF

```
# IP.upf (load_upf)
create_power_domain PDTop
# ... UPF commands
add_port_state VDDA
 -state "NORMAL $NORMAL_VOLT"
 -state {OFF off}
 # ... UPF commands
# SubSystem1.upf
create_power_domain PDSubSys1
set NORMAL_VOLT  1.0
load_upf IP.upf -scope subsys1/ipinst1
load_upf IP.upf -scope subsys1/ipinst2
# SubSystem2.upf
create_power_domain PDSubSys2
set NORMAL_VOLT  1.2
load_upf IP.upf -scope subsys2/ipinst1
load_upf IP.upf -scope subsys2/ipinst2
```

Power models allow creating parameters using add_parameter command. These parameters can be overridden with apply_power_model -parameters option. The add_parameter command is restricted to System Level power models in UPF 3.0. This has been extended to allow regular Tcl variables in UPF 3.1.

EXAMPLE: PARAMETERIZATION USING POWER MODEL

```
begin_power_model IP
 add_parameter NORMAL_VOLT -default 1.0
```

```
 create_power_domain PDTop
 # ... UPF commands
 add_port_state VDDA
  -state "NORMAL $NORMAL_VOLT"
  -state {OFF off}
 # ... UPF commands
end_power_model
# SubSystem1.upf
create_power_domain PDSubSys1
apply_power_model IP
 -parameters {
  {NORMAL_VOLT 1.0}
  }
# SubSystem2.upf
create_power_domain PDSubSys2
apply_power_model IP
 -parameters {
  {NORMAL_VOLT 1.2}
  }
```

C. File Management

The file-based mapping of power intent has led to the problem of management of a large number of UPF files corresponding to IPs. Since each IP needs to be specified in a separate file, the file-based approach is not scalable and causes various issues in real design environments. Any change in filename causes large changes in several `load_upf` commands unless variables are used. Not just the filename, even location of the file has an impact on the power intent and users must manually keep track of the location of IP UPF to ensure correct mapping with the IP instance.

Moreover, the limitation of an IP per UPF file limits the concept of a library of UPF files for hard IPs. This is different from Liberty libraries which are specified in a single file and are shipped by library vendors.

The following example highlights the impact of filenames and location in the `load_upf` command. The names of some hard macros UPF files can be cryptic and thus increase the chances of issues due to incorrect mapping.

EXAMPLE

```
load_upf /a/b/c/d/e/upf/sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00.upf
 -scope U_sms_wrapper1_int_top/U_sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00
```

Power models don't have this limitation. There can be multiple power models in the same file. The power models can be associated with the IP model using `-for` option. This becomes useful in large designs where changing the version of IP involves only updating the power model without any change in the `apply_power_model`.

EXAMPLE

```
begin_power_model SRAM_Synch_1Port
 -for {
   sadcls0c4l1p32x10m4b1w1c1p1d1r3s2sdz1rw00
   sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00
  }
```

With the power model approach, multiple models can be present in the same UPF file allowing the creation of a UPF library of power models, like liberty library or Verilog cell defines library, as shown in the below example.

Tools can introduce various mechanisms for the user to specify these libraries.

EXAMPLE

```
# SRAM Library
begin_power_model SRAM_Synch_1Port
 -for {
```

```
    sadcls0c4l1p32x10m4b1w1c1p1d1r3s2sdz1rw00
    sadcls0c4l1p32x10m4b1w0c0p0d0t0s2sdz1rw00
   }
  create_power_domain PDP
  create_power_domain PDA_SW -elements {uut} -scope uut
   # ... Other UPF Commands
end_power_model
begin_power_model SRAM_Synch_2Port
 -for {
    sadrls0c4l2p16x8m2b1w1c1p1d1r1s2z1rw00
   }
  create_power_domain PDP
  create_power_domain PDA_SW -elements {uut} -scope uut
   # ... Other UPF Commands
end_power_model
```

D. Verification vs Implementation mismatch

As UPF affects the entire flow, it is paramount that UPF simulated by an HDL simulator simulates the effect of UPF gate insertions by the implementation tool in the netlist. A mismatch between the isolation simulated by a simulator and isolation inserted by the logic synthesis tool can lead to a catastrophic failure if not fixed before the tape-out. The mismatch can be due to tool bugs or difference in UPF interpretation. A major source of UPF interpretation is the differences that creep in because implementation tool can implement the design in parts called sub-systems whereas the simulation tool will read the UPF for the entire design.

EXAMPLE

```
# IP UPF
create_supply_port VDD
create_supply_port VDD3
create_supply_port VSS
create_supply_set SSVDD  -function {{power VDD} {ground VSS}}
create_supply_set SSVDD3 -function {{power VDD3} {ground VSS}}
set_isolation ISO -domain PD -source SSVDD -sink SSVDD3
# SoC UPF
load_upf -scope IP1 ip.upf
connect_supply_net VDD -ports {IP1/VDD IP1/VDD3}
connect_supply_net VSS -ports {IP1/VSS}
load_upf -scope IP2 ip.upf
connect_supply_net VDD  -ports {IP2/VDD}
connect_supply_net VDD3 -ports {IP2/VDD3}
connect_supply_net VSS  -ports {IP2/VSS}
```

In the above scenario, since verification tool will see the entire UPF, for IP1 instance, SSDD and SSDD3 become equivalent and therefore isolation strategy ISO will not apply on any ports. For IP2 instance, isolation strategy ISO will apply on the ports. An implementation tool will however always implement for the IP isolation due to isolation strategy.

The mismatch between verification and implementation tools is largely due to separate compilation and application of UPF in implementation flow. In contrast, a typical verification flow compiles the UPFs of all the integrated IPs together.

With power model and terminal boundary semantics, the verification tools can mimic the separate compilation of the IP UPF. The terminal boundary semantics were introduced in IEEE UPF 1801 2015 to consistently define restrictions on modification of UPF for a macro model from outside. This semantics are enabled for Hard and Soft macros.

VI. LIMITATIONS IN UPF 3.0

A. *Protected Environment*

The `begin_power_model/end_power_model` commands do not provide protection from the global variables. So, any global variable defined outside can be accessed within the model. Also, any global variables modified within the power model can unintentionally modify the UPF code outside the `end_power_model`. To address the limitation, a new command `define_power_model` has been introduced in UPF 3.1. This command provides a protected environment and has the syntax is like Tcl proc.

SYNTAX

```
define_power_model power_model_name [-for model_list] {
    UPF_commands
}
```

EXAMPLE

```
define_power_model upf_model_param -for cellA {
  add_parameter PD_NAME -default PD_MODEL
  create_power_domain $PD_NAME -elements {.} -supply {ssh1} -supply {ssh2}
  # other commands ...
}
...
apply_power_model upf_model_param -parameters {{PD_NAME PD2}}
```

B. *Supply and Logic Port Connection*

UPF 3.0 only allows associating the supply sets with `apply_power_model` command. In some cases, it is needed to connect the logic and supply ports to the top-level supply ports. In such case, users must explicitly write `connect_logic_net` and `connect_supply_net` commands for each instance ports. This can become tedious and error-prone for power models with many instances. Acknowledging this limitation, the `apply_power_model` command is extended to have a new option `-port_map` which allows connecting logic and supply nets in the command itself.

EXAMPLE

```
apply_power_model upf_model
 -elements {I1 I2}
 -supply_map {{PD.ssh1 ss1} {PD.ssh2 ss2}}
 -port_map {{isolate_n iso}}
```

C. *Tcl parameters in add_parameter*

The `add_parameter` command in UPF 3.0 is only used to specify constants for System-Level power models. This has been extended to allow Tcl parameters which are special Tcl variables which can be used in UPF file to provide parameterized UPF.

EXAMPLE

```
define_power_model upf_model_param -for cellA {
 add_parameter PD_NAME -default PD_MODEL
 create_power_domain $PD_NAME -elements {.} -supply {ssh1} -supply {ssh2}
 # other commands ...
}
apply_power_model upf_model_param -parameters {{PD_NAME PD2}}
```

VII. FUTURE EXTENSIONS

A. *Implicit instantiation*

1) Top Design

Most EDA tools accept a single UPF file which applies to one top-level model. This limits the verification environments that allow multiple power managed design top models. For example:

- Spice netlist instantiating multiple digital models for which the UPF is available
- A system environment instantiating multiple SoCs. This is currently achieved by writing a UPF for testbench which has `load_upf` for all the SoCs instances.

In such cases, the environment already handles the connection to the top models and hence there is no need for any UPF specification which provides the connections. For such cases, the power model for the top level can be implicitly applied. This semantics are currently not defined in the UPF 3.0. We have filed an enhancement request for this in the IEEE 1801 WG for considering it in the next revision.

2) Hard Macro

Hard macros typically have a Liberty file which contains the interface of the macro including the supply pins. They can have UPF power models which augment the power information present in the Liberty specification, e.g. power states or fine-grained power switches. In very large designs, which has thousands of macros, manually instantiating such UPF files can be tedious and error-prone. Moreover, in some cases, there are no explicit connections needed for the supply pins as it can be inferred from the power domains in which they are instantiated. For these scenarios, the power model corresponding to the macro model can be implicitly applied. The current version of UPF doesn't allow this capability. We have filed an enhancement request with the IEEE 1801 Workgroup to add the capability.

B. Updating the power model

A UPF specification is designed to be used at different stages in the design flow, i.e. RTL to Gate Level. The subsequent stages may add new information in the design which may require additions to the UPF specification. The commands in UPF have special semantics called Successive Refinement which allows updating existing UPF objects with more details when more information is available. This is achieved via `-update` option to some of the UPF commands. However, this capability is not available for power models. We have filed an enhancement request in the IEEE 1801WG for extensions to the new revision.

VIII. CONCLUSION

In this paper, we have proposed a model-based power intent specification for UPF using power models which works seamlessly with the hierarchical-based methodology for modern designs. This methodology uses semantics defined in the UPF specification to eliminate various issues in UPF processing. This methodology will empower users to specify the power intent with minimal code and rely on inbuilt semantics to minimize errors. It will also empower tools to improve the performance of compilation as they can apply better optimization leveraging the model-based semantics of power models. The style fits well not only with IP based flow methodology but also solves the problem for library vendors that provide UPF for their cells or macros. The library vendors can now generate one UPF library with all UPF models in one file like liberty file. This was not possible with `load_upf` approach that required one UPF file to be provided for every cell.

REFERENCES

[1] IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems," in IEEE Std 1801-2015 (Revision of IEEE Std 1801-2013) , vol., no., pp.1-515, March 25 2016

[2] Mohit Jain, Amit Singh, JSSS Bharath, Amit Srivastava, Bharti Jain , "Power Aware Models: Overcoming barriers in Power Aware Simulation", DVCon Europe 2014.

[3] Tim Kogel, "Applying UPF 3.0 for Early, System-level Power Analysis of SoCs with DDR Memories", DVCON EUROPE 2016.

[4] VCS Native Low Power with MVSIM, https://www.synopsys.com/verification/simulation/vcs-with-mvsim.htm