Refining Successive Refinement:

Improving a Methodology for Incremental Specification of Power Intent

Desinghu PS, ARM Ltd., Sheffield, UK (*desinghu.ps@arm.com*)

Adnan Khan, ARM Ltd., Cambridge, UK (*adnan.khan@arm.com*)

Erich Marschner, Mentor Graphics, Ellicott City, MD, USA (erich_marschner@mentor.com)

Gabriel Chidolue, Mentor Graphics, Newbury, United Kingdom (*gabriel_chidolue@mentor.com*)

***Abstract*—"Successive Refinement" is a methodology for the use of IEEE Std 1801™, the Unified Power Format (UPF), to specify power intent—i.e., the power management structures that are intended to be used to control power in a low-power design. This methodology enables incremental design and verification of the power management architecture, and it is specifically designed to support specification of power management requirements for IP components used in a low power design. The methodology is still relatively new and continues to evolve. In this paper, we present lessons learned from a recent application of Successive Refinement to specify power management requirements for a complex SoC and verify the power management configuration of that system. We present a revised definition of Successive Refinement that is better tuned to the needs of IP vendors and is expected to work more reliably in a multi-vendor tool flow.**

Keywords—low power design;upf; power aware verification

I. Introduction

Power management is a fundamental requirement in today's electronic systems. IEEE Std 1801 UPF [1] provides a method of specifying how power will be managed in a given design. This power intent specification is separate from the functional specification for the design, which is typically given in synthesizable RTL code in SystemVerilog or VHDL. However, information about power management does not all become available at the same time, and therefore it is necessary to adopt a methodology that allows for incremental specification of power intent as the design evolves.

Successive Refinement is a methodology that supports such incremental specification. Originally defined as part of the IEEE 1801™-2009 UPF specification (also known as UPF 2.0), this methodology has been further elaborated in later revisions of the standard [2] and is now being used in production flows. A previous paper [3] presented a detailed overview of Successive Refinement and its advantages.

Support for Successive Refinement has continued to evolve as the UPF standard has been updated and as EDA tools' support for the UPF standard has become more complete. In this paper we present lessons learned from recent application of Successive Refinement to the design and verification of a complex SoC. We also present a revised definition of the Successive Refinement methodology that incorporates changes to address certain problems encountered in that application and factors recent semantic changes in the latest version of the UPF standard.

II. Background

A. *Successive Refinement Flow*

In a typical design flow for a complex System on Chip (SoC), the system is constructed using many IP components that are integrated together to provide the functionality required. Use of IP is essential in today's design flows in order to meet time-to-market requirements and leverage existing technology most effectively. Designing the power management mechanisms for such a product thus necessarily involves consideration of IP requirements as well as system integration concerns.

Successive Refinement addresses both of these issues. First, the methodology defines how an IP provider can use UPF to specify constraints on the use of an IP component within a system, without knowledge of the characteristics of the system. This enables an IP provider to deliver such constraints along with the RTL functional specification of the component, to ensure that the component will be used correctly in any given application. Second, the methodology defines how UPF can be used by the system integrator to specify the logical configuration of power management for the individual IP components used in the system and for the system as a whole. This enables early verification of the power management architecture before any implementation decisions are made. Finally, the methodology defines how UPF can be used by the system implementer to realize the power intent in the context of a given technology and implementation approach.

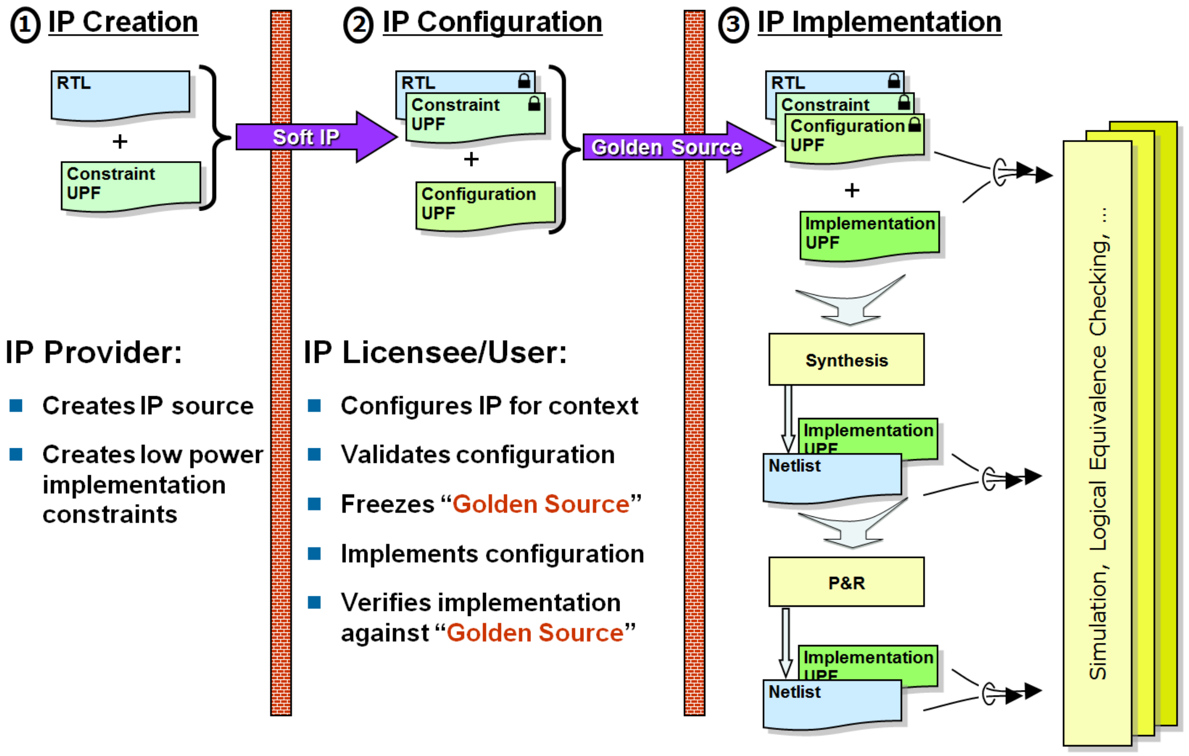

Figure 1. The Successive Refinement Flow

Successive refinement is illustrated in Figure 1. The IP developer creates Constraint UPF that accompanies the RTL source for a soft IP component. The IP consumer (system integrator) adds Configuration UPF describing his system design including how all the IP blocks in the system are configured and power managed. The Configuration UPF is loaded together with the constraint UPF for each IP so that tools can check that the constraints are met in the power management configuration for the design. The RTL plus the constraint and configuration UPF is referred to as the "golden source". Once the configuration specified by the golden source has been validated, the implementer adds Implementation UPF to specify implementation details and technology mapping. The complete UPF specification then drives the implementation process.

B. Benefits of Successive Refinement

Successive Refinement provides two major benefits. First, it enables an IP provider to specify how an IP block can be used in a low power system, in such a manner that tools can check statically to ensure that each instance of a given IP in a system satisfies its usage requirements. This boosts productivity for the end user by helping to identify any usage issues very early, through static analysis, rather than waiting until the necessary infrastructure is present to run simulations. Second, it enables system designers to separate the logical functionality of power management for a system from the technology-specific implementation of the system. This enables dynamic verification to start earlier than otherwise, helps make debugging easier by enabling separation of concerns in different simulation runs, and enables the user to preserve verification equity when retargeting a design to a new technology.

III. Applying Successive Refinement to a Complex System on Chip

The Successive Refinement methodology described in [2] was applied in the design and implementation of a System on Chip (SoC) that consists of complex IPs such as multi-core processors, graphic engines, interconnect subsystems and memory controllers. The design also included distributed power management logic embedded in various levels of the design hierarchy. Figure 2 shows a scaled down representation of the design that was implemented.

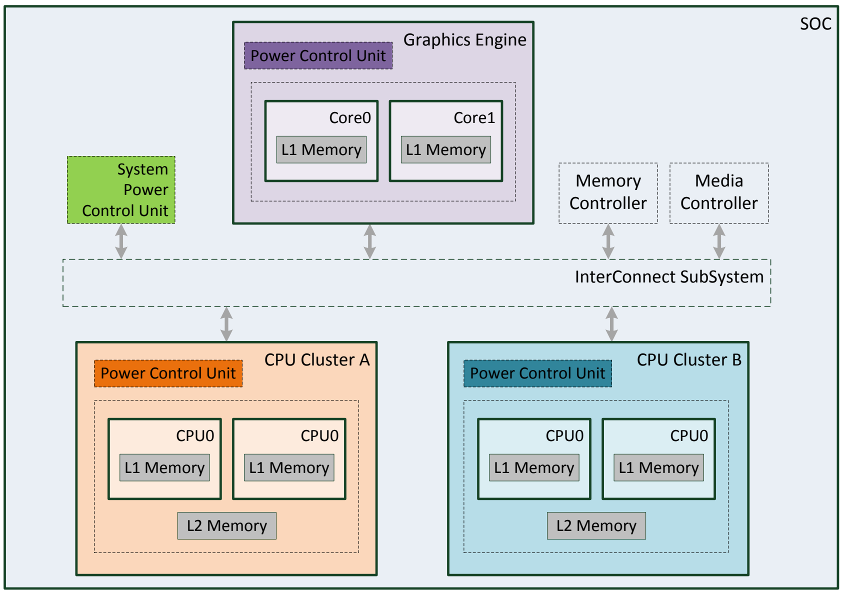

Figure 2. SoC block diagram

The physical implementation of the SoC was done hierarchically with a bottom-up approach in which portions of the design were implemented separately as hard macros and then integrated with the rest of the design for further verification and implementation. Each hard macro created in this process required its own standalone constraint, configuration and implementation UPF. In the SoC represented by Figure 2, CPU cluster A, CPU cluster B and the graphics engine were hardened independently with their own standalone power intent.

IV. Practical Challenges In Applying Successive Refinement

In abstract, Successive Refinement is straightforward: constraints are design specific and specified for IP blocks; a system built from IP blocks is configured for power management and is application specific; implementation details are technology specific provided to guide the implementation of the system. However, in practice, the implementation flow is often done in incremental stages, due to tool capacity limitations, IP reuse requirements, layout requirements, schedule needs, or other factors. In this section we discuss some of the practical challenges that we have encountered in applying Successive Refinement in the design and implementation of a complex SoC.

A. Handling Hard Macros

For smaller designs that can be implemented in one step, the Successive Refinement methodology is relatively easy to apply. Constraint UPF available for IP blocks can be loaded along with the configuration UPF for the whole system, static verification can be used to confirm that the IP constraints are satisfied by the configuration UPF, and then the technology-independent, logical power management architecture defined by the configuration UPF can then be functionally verified in simulation. Later, implementation UPF can be added to drive the implementation process, static verification can again be used to verify that the implementation details do not violate any requirements of the logical power management architecture, and then the technology-dependent implementation of power management can be functionally verified in simulation.

Large SoC implementation may also be done concurrently as well as incrementally, by independently implementing certain sub-blocks as hard macros that can then be integrated back into the rest of the design. These partial implementation steps result in a mixture of technology-dependent UPF representing power intent that has been (or will soon be) implemented within a part of the design, and technology-independent UPF representing power intent that is not yet implemented nor ready for implementation. This creates a requirement for careful handling of the interface between the hardened blocks and the still soft context in which those blocks are instantiated.

There are two cases of hardened blocks to consider. One case is the traditional *hard macro*, exemplified by a memory macro that exists as a library element and is modeled with an HDL behavioral model for simulation and a Liberty model for implementation. The behavioral model may be power aware—i.e., may include the power supply ports required for the implementation—or power unaware. The Liberty model will typically include definitions of those supply pins (pg_pins) and other related information to make it power aware. In this case, since the behavioral model and/or the Liberty model is already power aware, there may be no UPF power intent specification at all. If there is a UPF power intent specification, it can only define characteristics of the interface to such a model, since the HDL model is strictly behavioral and does not represent the internal structure of the macro. Since the UPF for this macro can only define interface characteristics, the UPF power intent of the context in which it is instantiated can only refer to interface characteristics of the macro instance.

The other case is a *hardened soft macro,* which involves an RTL subtree of the design hierarchy (e.g., a component that is instantiated in the design) that has been prepared for implementation by adding implementation UPF to the constraint and configuration UPF that applies to that subtree. In this case, the synthesizable RTL is still being used as the simulation model, and a Liberty model for this block probably does not exist. In this case, the UPF power intent describes the power management architecture within the RTL subtree—the power domain partitioning, the strategies for isolation, level shifting, retention, and repeater insertion, and the implementation of power management and power distribution logic. However, since this RTL block will eventually be replaced with a hard macro, the UPF power intent for the internal implementation of this subtree should not be referenced by UPF power intent for the system as a whole. In effect, there should be a hard boundary between the hardened soft macro and the context in which it is instantiated. This hard macro boundary includes UPF-defined supply ports and logical control ports that serve as connection points between the outer environment and the internal logic of the macro. Here again, the power intent for the external environment should not refer to any internal structures within the hardened soft macro, even though the UPF does specify internal details; the external power intent should only refer to characteristics of the now hardened interface.

In either case, a hardened block will have explicit supply ports on its interface. For a hard macro that is initially modeled with a power unaware simulation model, the instantiating context must eventually be adapted to connect to the explicitly supply pins defined in the Liberty model. This may involve adding some implementation-dependent information into the configuration UPF for that outer context. For a soft macro that is hardened in preparation for separate implementation, this may also involve restructuring the UPF of the context in order to create the hard boundary required for separate implementation.

A hardened block may also need to export information about its internal state, especially if it contains embedded switches that generate switched supplies used by internal domains within the macro or used to power driving logic of macro outputs or receiving logic of macro inputs. Representing this internal state information is another challenge that needs to be addressed by the Successive Refinement methodology.

B. Isolating UPF-created Power Control Signals

When a hardened soft macro instance is present in a larger context, and that larger context is prepared for its own implementation, the UPF-defined logic (control) ports on the inner macro may require isolation. This leads to a need for specifying isolation strategies on UPF-defined logic ports of the inner macro in the UPF for the larger context. Such ports will typically be on the lower boundary of the power domains of the larger context.

UPF 2.1 does allow isolation strategies to apply to UPF-created logic ports. However, not all tools today support this capability. There is also an issue in the UPF 2.1 standard with specifying clamp value constraints on such ports, to enable static checking of the power management configuration. While the UPF 2.1 spec says that clamp value attributes can be specified for UPF-created logic ports, it also says that all **create_logic_port** commands are effectively executed *after* all **set_port_attribute** commands, which appears to make such usage impossible.

C. Defining Power States Effectively

The Successive Refinement methodology attempts to cleanly separate technology-independent logical configuration aspects and technology-dependent physical implementation aspects of power intent, to enable early verification of the logical configuration and reuse of that logical configuration for different target implementation technologies. This separation of configuration and implementation concerns has a particular impact on the definition of power states, especially those for supply sets, since the power state of supply set can be defined with both a logic expression (representing the control conditions under which that power state will occur) and a supply expression (representing the actual supply state and voltage values that will exist when the supply set is in that state).

Ideally, logical control conditions would be associated only with power states of power domains and not with power states of supply sets. This would allow configuration UPF to completely avoid having to specify supply sets and their power states. However, since UPF 2.1 only allows the –simstate specifications (which determine simulation semantics) on supply set power states, this is not possible in UPF 2.1.

Another potential issue related to supply set power state definitions is the potential for redundancy between the definition of power states of the primary supply set of a power domain and the power states of the domain itself. For example, a control input such as "SLEEP" might be referenced in the definition of both a power domain PD, in which it is treated as an abstract mode control, and in the definition of a power state of the primary supply of PD, in which it is treated as a switch control signal. Such redundancy creates a multiple-update failure mode, in which one of the two definitions might be updated while the other is not. In general, we want to avoid this sort of redundancy to minimize the risk of such failures.

V. Recommended Solutions for These Challenges

In this section, we present an overview of the methods we adopted to address the above-mentioned challenges.

A. Defining Power States

We had three goals to achieve in specifying power states for domains and supply sets:

1. Avoid redundancy in the specification
2. Separate configuration and implementation concerns
3. Ensure clean composition of power states over the entire design

We adopted an elegant way of representing power states of supply sets and power domains that achieves all three goals.

To avoid redundancy in the specification, we chose to define domain power states in terms of the power states of the relevant supply sets for that domain. This avoids creating a multiple-update problem that would exist if we were to reference the same control signals in both the domain power state definitions and the supply set power state definitions. Instead, this approach reflects the dependency of domain power states on the power states of the domain's supply sets, as well as on power states of subordinate domains in the system.

Separation of configuration and implementation concerns is only an issue for power states of supply sets, which can be defined with both a logic expression and a supply expression. One approach we considered involved defining power states of a supply set only in terms of the supply state of its functions in configuration UPF - for example, using the supply expression {power==FULL_ON && ground==FULL_ON} to define the power state ON of a supply set. The voltage information could then be specified in the implementation UPF, once the target technology is known.

This is possible in UPF using the -update option for add_power_state, which combines a new definition of a power state with a previous definition of the same state. For example,

```
add_power_state -supply PD.primary \
    -state {ON -supply_expr {FULL_ON}}
```
and a later update command
```
add_power_state -supply PD.primary -update \
    -state {ON -supply_expr {FULL_ON 1.0}}
```
is defined to be equivalent to
```
add_power_state -supply PD.primary \
    -state {ON -supply_expr {{FULL_ON} && {FULL_ON 1.0}}}
```
which is semantically equivalent to
```
add_power_state -supply PD.primary \
    -state {ON -supply_expr {FULL_ON 1.0}}
```
However, not all tools support this feature of UPF, so we found that it is best not to use this mechanism.

Instead, we chose to define supply set power states using only a logic expression in configuration UPF and adding the whole supply expression - specifying both supply state and supply voltage - in the implementation UPF as shown below.

Configuration UPF:
```
add_power_state PD.primary \
    -state {ON -logic_expr {sw_ctrl==1}}
```
Implementation UPF:
```
add_power_state -supply PD.primary -update \
    -state {ON -supply_expr {FULL_ON 1.0}}
```

This enabled verification of the logical configuration by using control signals to control the power states and therefore simstates of the supply sets upon which power domains depend. Later, when the supply expression had been added in the implementation UPF, verification could be done using the supply information also.

We also adopted the following rules with regard to the objects that could be referenced in the logic expressions used to define domain and supply set power states:

1. The logic expression for a power state of a supply set shall be defined only in terms of logic ports, logic nets, interval functions, and/or power states of the given supply set or supply set handle.
2. The logic expression for a power state of a power domain shall be defined only in terms of power states of supply sets or supply set handles, and/or power states of lower level power domains.

This ensures clean composition of the power states over the whole design, in a bottom-up fashion, so that the power states of leaf-level objects and legal combinations of those states ultimately determine the legal states of the system.

B. *Creating and Integrating Hard Macros*

We adopted a methodical approach to creation and integration of hard macros in this project.

Physically *hard macros* such as memories and PHYs were integrated by creating power models. Power models for each *hard macro* were created using a Tcl procedure consisting of UPF commands. The power model was used to supplement the liberty model of the *hard macro*. The supply ports and port attributes were imported from the liberty model by EDA tool. The power model were used to define power states and set the boundary conditions of the macro in the context of the higher domain. The internal supplies and switches in the *hard macro* were modelled and the power state of the domain was defined in terms of power states of internal supply states. The explicit supply ports of the *hard macro* were associated into the supply sets within the memory power model. An abstract of memory power model is shown here.

```
proc memory_power_model { pd_name mem_instance sw_ctrl ret_ctrl } {
  create_power_domain $pd_name -elements $mem_instance
```

<UPF commands to create additional supply set handles based on model requirements>

<Associate supply ports of the hard macro to the supply set handles within>

<Model the internal switches states>

```
    add_power_state $pd_name.primary -update \
      -state {ON  -logic_expr {on_logic_condition} \
                  -supply_expr {on_supply_expr} } \
      -state {OFF -logic_expr {off_logic_condition} \
                  -supply_expr {off_supply_expr} }
    add_power_state $pd_name \
      -state {ON  -logic_expr {on_condition}} \
      -state {RET -logic_expr {ret_condition}} \
      -state {OFF -logic_expr {off_condition}}

    <UPF commands to model boundary conditions of the hard macro using set_port_attributes>
  }
```

Power models of each *hard macro* are loaded in the parent context by calling the Tcl procedure. The power state of the parent domain in updated with the state of the *hard macro*.

```
  memory_power_model PDMEM <memory_instance_path> sw_ctrl  ret_ctrl
  add_power_state PD -update \
    -state {ON  -logic_expr {PDMEM != OFF}} \
    -state {RET -logic_expr {PDMEM == RET}} \
    -state {OFF -logic_expr {PDMEM == OFF}}
```

In cases where the number of memory instances was large, an abstract power model over a collection of *hard macro* instances was created to minimize the number of domain and models.

Supply net connections to the supply port of the *hard macro* in the implementation UPF ensured proper association of supplies from the top all the way to the *hard macro* seamlessly.

The physical implementation of the SoC was done bottom-up and this required changes to the interface context and hardening the boundaries of the hardened soft macro. The hardening process introduced supply ports on the boundary of the macro and hence the supply propagation needed to be done through supply ports and supply nets in UPF2.x rather than just via supply set associations.

When an RTL component within the system was hardened, the context of the hardened macro needed to be adapted in the higher level constraint/configuration UPF. The isolation/level-shifter requirements needed to be suitably adapted in the higher level based on the macro implementation.

The interface description of the macro also needed to change when the macro was used in the context of the higher level configuration.

A standalone and complete power intent for the hardened soft macro was specified with its own constraint, configuration and implementation UPF used to verify and implement the soft macro. The UPF of the soft macro were loaded in UPF of the higher level module to model the power intent of the hardened soft macro boundary ignoring the internals of the hardened soft macro. The boundary conditions of the hardened soft macro were defined in the implementation UPF of the soft macro using environment variables that would define the context of the UPF. A snippet below shows the use of environment variable to model the boundary condition.

```
  if {$env(CORE_UPF)==1 && $env(TOP_UPF)==1} then {
    set_port_attributes -ports \
      [find_objects . -pattern * -object_type port -direction in] \
      -exclude_ports "sw_ctrl iso_ctrl ret_ctrl" -driver_supply PD.primary
    set_port_attributes -ports \
      [find_objects . -pattern * -object_type port -direction out] \
      -receiver_supply PD.primary
    set_port_attributes -ports "sw_ctrl iso_ctrl ret_ctrl" \
      -driver_supply PD.aon
  } elseif {$env(CORE_UPF)==0} then {
    set_port_attributes -ports \
      [find_objects . -pattern * -object_type port -direction out] \
      -driver_supply PD.primary
    set_port_attributes -ports "sw_ctrl iso_ctrl ret_ctrl"
      -receiver_supply PD.aon }
```

C. *Powering isolation and retention cells*

We decided to avoid use of the supply sets default_isolation and default_retention that are predefined for each power domain. The use of default_isolation and default_retention supply set handles of the power domain complicates and confuses the isolation/retention policies. In particular, the default_isolation supply set presupposes that dual-rail isolation cells requiring backup power will be used for isolation, yet this is an implementation decision that need not be made in configuration UPF. Similarly use of the default_retention supply set tends to suggest that retention will be implemented using balloon latches, whereas the implementation may choose to use live-slave latches instead.

VI. Results of this Work

A UPF power intent specification for a SoC with multiple IPs having different levels of physical hierarchical implementation was created such that each physical implementation hard macro has its own UPF specification, and each UPF specification is defined based on Successive Refinement. The UPF specifications for individual IPs were verified for structural checks through static checks and formal methods.

The power models created for memories and PHY were elegant and re-use of the power model was achieved in the successive refinement process. Modelling the power intent of hard macros in the design with power states for the hard macro defined in the power model was key in successful description of the power intent of the design.

The higher level interface of each IP was modelled through port attributes. The port attributes were set based on environment variables that defined the UPF interface scenario – IP or system level. Thus structurally clean IP UPFs were integrated at the sub-system top. The structural checks at top level were easier because by the time they were run, each of the IP's UPF specifications had been verified to be structurally correct.

To perform Low power verification, we could verify each of the IP with the UPF in their block level verification environment and at the top level using the SoC level verification environment. The use of UPF for IP block verification, IP *hard macro* implementation, SoC verification and SoC implementation was seamless with no changes to the IP or SoC UPF between processes.

The power state composition as a result of this improved successive refinement was clean. A functional coverage model for the UPF was built and 100% coverage achieved over different power states, state transitions of the power domains and *hard macros*.

References

[1] IEEE Standard for Design and Verification of Low-Power Integrated Circuits, IEEE Std 1801™-2013, 6 March 2013.

[2] IEEE Standard for Design and Verification of Low-Power Integrated Circuits—Amendment 1, IEEE Std 1801a™-2014, August 2014.

[3] A. Khan, J. Biggs, E. Quigley, and E. Marschner, "Successive Refinement: A methodology for incremental specification of power intent", DVCon 2015, March 2015.