Verification Reuse for a Non-Transaction Based Design across Multiple Platforms

Luis Li, Pablo Salazar, Andrés Cordero
Hewlett Packard Enterprise, American Free Zone Building C-8, Heredia, Costa Rica 40103

***Abstract*- In modern technology, an era of digital disruption has arrived where Big Data, Internet of Things (IoT) and mobility are radically transforming the businesses of applications and data. High End Servers are prone to this change and their architecture has evolved into more complex structures. At Hewlett-Packard Enterprise *(*HPE), a quick response is essential to provide a server portfolio composed of multiple products and solutions which enable HPE to meet our customer's unique and intricate needs. The verification of multiple digital systems triggers a tremendous need for code reuse that conserve verification efforts. Moreover, the large percentage of our design that is non-transaction based has created the requirement to develop a verification environment that is highly configurable to adapt to those conditions. This paper will explore reuse and configuration techniques applied to develop a verification environment capable of managing non-transaction based stimuli injection and to provide a tool to administer checking and coverage over multiple platforms. The final section highlights the results of this methodology.**

I. Introduction

A. Non-Transaction and Multi-Platform Designs

A verification engineer's main role involves continually interacting with digital communication protocols and standards which is an essential task since it is the fundamental way devices communicate with each other. As such, any digital system can be defined as a device, from the most complex Application-Specific Integrated Circuit (ASIC) such as a last-generation microprocessor to a simple Universal Asynchronous Receiver/Transmitter (UART) interface or to a Serial Peripheral Interface (SPI) converter implemented in a Field Programmable Gate Array (FPGA).

In design verification, the most basic or fundamental piece of information in digital protocol is deemed a "transaction." During the verification process of a design, the environment generates and injects sets of transactions to establish communication with a design and then performs response checking to confirm that this Design Under Test (DUT) is behaving as expected. Currently, an abundance of protocols exist in the electronics industry; therefore, standardized verification methodologies like the Universal Verification Methodology (UVM) are based on the assumption that most verifications environments are built to support one or more communication protocols. In fact, many verification Intellectual Property (IP/VIP) solutions offered by Electronic Design Automation (EDA) vendors are available in the open market if internal project schedules are too stretched and manpower limited to create a custom solution.

To illustrate, what happens when a verification engineer determines that a device's pinout is not following a specific protocol and rather is merely a collection of bit-wise (ON/OFF) signals? How can the engineer utilize a verification methodology like UVM to build a non-transaction based environment? To make the challenge even more difficult, what if the environment must support not only one DUT but several similar ones (platforms)? In the detailed analysis following, an answer to these questions is provided.

DUT pinout values in a transaction based verification environment are constrained by the protocol itself as the injection must respect timing diagrams identified and described in the standard. However, the verification environment complexity grows exponentially from transaction based to non-transaction based design. For a non-transaction case, pinout activity may arise at any time and in any order; therefore the state-space grows exponentially as a function of the number of DUT inputs and outputs. Typical examples of non-transaction based designs are as such: a controller of an industrial machine that responds to several switches and sensors, a power sequencer for a server programmed on a Complex Programmable Logic Device (CPLD) or FPGA, and any device with several state machines (SM) and/or sequential logic in its design.

B. Case Study Overview

At HPE, design engineers interact with digital systems that meets both non-transaction based and multi-platform conditions. In the most recent generations of the ProLiant Server Family, a programmable device known as the System CPLD (or CPLD) will be our case study throughout this paper.

Inside of each server, the CPLD directs the hardware monitoring and administrative functions specifically focusing on server power management. It also governs reset, fault detection, glue logic, and other important features inside the server. In addition, it interacts with the Integrated Lights Out (iLO) management ASIC, the Southbridge Chipset (SBC), processors, power supply units and other devices. This hardware architecture generates a large number of asynchronous signals converging in the CPLD which in turn generates numerous functional scenarios. Those signals have a discrete nature in the sense that they are used to represent ON/OFF signaling and they do not follow a specific standard protocol, which turns out to represent the non-transaction based nature

of the CPLD. Additionally, the number and kind of devices connected to the system CPLD differ from one server to another thereby making it a multi-platform system as well.

C. Solution Overview

To create a verification environment for a non-transaction based DUT and additionally ensure the ability to support several platforms, it is necessary to implement verification code reuse and highly configurable components in a layered structure. Over a period of years, the HPE team developed a flexible environment useful for 15+ CPLD unique platforms using the UVM methodology in SystemVerilog (SV). The resulting analysis demonstrates the creation of an environment that is highly adaptable to several CPLD configurations as well as able to manage a non-transaction based DUT. The technical challenge of commanding numerous platforms with unique hardware requirements while simultaneously achieving a high level of reuse required through the implementation of configuration files will also be examined.

The standard method in checking discrete signals utilizes SystemVerilog Assertions (SVA); however, our analysis will demonstrate why SVAs were not entirely suitable for this particular project. Exploration of the solution implemented to analyze and correlate the discrete data resulted in the creation of a UVM-based tool that provides verification engineers all the information collected and provides the possibility to easily make signal evaluations and combinations for certifying different scenarios. This newly created UVM tool provides in its entirety the functions and components required to build a complete scoreboard. The case study will also establish how this tool can be used via a micro language designed to simplify the scoreboards' writing and reading and accelerate their development.

II. Case Study

This paper explains how an UVM/SV verification environment was architected and structured in order to inject stimuli and check behavior of the CPLD included on each HPE ProLiant server. The design implemented in this CPLD meets the conditions of both non-transaction based and multi-platform designs.

The following are the basic assumptions regarding the CPLD:

- **Multi-platform**: ProLiant is a family of servers each having its own specialized CPLD with approximately 90% of the design for each platform sharing common code and the remaining 10% using custom-made code to the specific features of the server. For each generation of the ProLiant server, many different server variations can co-exist with each requiring verification.
- **Non-transaction based**: Most of the pinout in our CPLD environment has an ON/OFF nature, meaning that a majority of the signaling processed by the CPLD is unresponsive to a specific communication protocol; instead, the signaling processed is used for providing the status of specific devices linked to the server or are used to control specific sequencings performed by the CPLD.
- **Sequencing as its main task**: Several state machines (SM) populate the CPLD's Register-Transfer Level (RTL) in order to perform tasks such as server' power sequencing, LED control, server monitoring, and fault/error detection.
- **Support for several server structural devices**: Even in scenarios where the CPLD functionality maintains consistency among various platforms, the devices linked to the server can be vary drastically in type and in quantity. Examples as such include: the number and model of CPUs, Voltage Regulators Modules (VRMs), power supplies, south bridge, mezzanine cards, power supplies, hard drives, and others.
- **iLO partnership**: HPE's iLO is the HPE server provisioning and management ASIC. It simplifies server setup, provides access to server health information, enables server management at scale, improves server power and thermal control and conveys basic remote administration [1]. The iLO ASIC and CPLD are co-dependent in the sense the they partner with each other for their server control and monitoring tasks. This is because the iLO has access to a majority of the high-speed buses in the server whereas the CPLD has access to the more discrete/customizable features of the server.

III. Solution

Our challenge was bifurcated as first, UVM items, components and sequences must be adapted to a non-transaction nature, and second, the solution necessarily must support multiple CPLD platforms while being cost efficient. One possible solution would be to develop a single verification environment per platform; however, this precludes cost efficiency as development workforce and maintenance costs increase exponentially. Thus, the solution pursued takes advantage of the following facts:

- Ports in the DUT top module are ON/OFF and therefore are controlled and monitored consistently and similarly allowing reused read and write methods.
- Common code permeates the DUT which allows verification code reuse. Differences arising are patched via configuration in generic components.
- Ports in the DUT top module may be easily grouped together if they are part of the same functionality or are connected to the same peripheral device.
- CPLD functionalities are highly similar thus providing uniformity across high level platforms.
- Behavioral checks and coverage collection evolve with common code to ensure maintenance efficiency. This discards the use of concurrent SVAs as they are prohibited within classes and can only be written in modules and SystemVerilog interfaces [2].
- Code reuse, configuration and abstraction layers are the quintessential elements in our solution.

The CPLD verification environment provides support to all platforms of a specific ProLiant generation. To effectively administer differences among platforms, the verification environment is structured in layers. The lowest layer encompasses the most generic

and highly configurable features of the environment while the top most layer incorporates functional aspects shared by all platforms. The overarching goal is to provide easy platform integration tools to enable users to focus primarily on the functional verification and alleviate or mitigate differences at the lower level. The high level verification environment provides a set of common methods that can be used to write sequences and test-cases in a sustainable manner, empower scoreboarding, and support coverage collection tools.

To illustrate, Figure 1 presents the three abstractions layers which compose the CPLD verification environment. Moving from the physical to the functional layer increases the amount of code accordingly. The physical layer contains extremely generic components with small pieces of code but those generic components are highly configurable. Next, the device specific layer comprised of a library of device specific behaviors that can be selected complementing the platform needs. Finally the major part of the infrastructure code exists at the functional layer where the components are been re-used or shared across a majority of platforms. The layers are discussed in greater depth in the following sections.

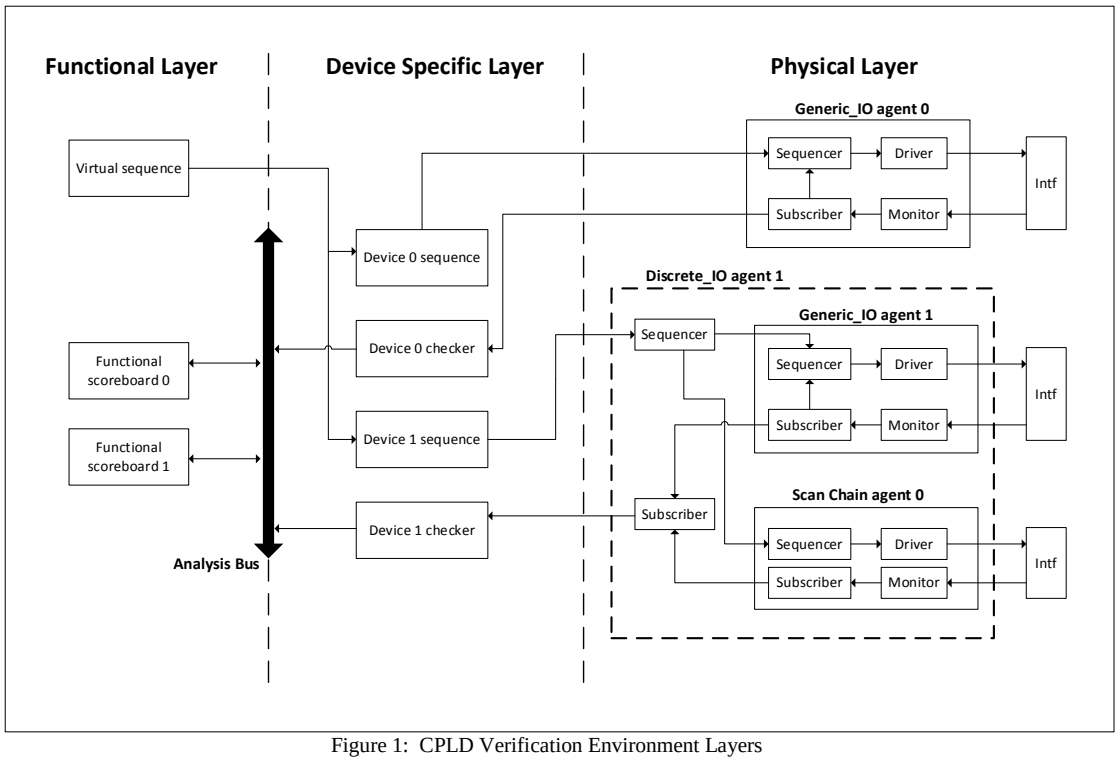

Figure 1: CPLD Verification Environment Layers

A. Physical Layer

The architecture of the CPLD connecting to other devices in the server can vary significantly from one platform to another. Some servers are comprised of a simple structure such as one CPU, no power supplies, and a small number of peripheral devices. Other servers offer more complexity of up to four CPUs, multiple mezzanine cards, two power supplies, etc. Platform designers and server designers together establish these connections.

In design, Scan Chains are commonly incorporated in order to save some pinouts or to increase CPLD connection width. Normally non-critical monitoring and controlling signals are linked with the CPLD via scan chains. To clarify, Figure 2 illustrates an example of a device (Device 0) connected to the CPLD using single bit inputs and outputs (Generic I/O). Another device (Device 1) is coupled to the CPLD utilizing a mix of Generic I/Os as well as through scan chain 0 and 1. One device can use multiple scan chains to transmit/receive bits while a specific scan chain may be used to transmit/receive bits from multiple devices.

The physical layer must provide highly configurable components with small amounts of code. In order to meet this condition, the verification environment composition includes the following agent packages at this layer.

- **Generic I/O**: Supports an infinite number of bit-wise inputs, outputs and inouts mapped directly to the CPLD pinout.
- **Scan Chain**: Supports a static number of bit-wise CPLD inputs and/or outputs mapped via scan chain in which the width of each scan chain offers a platform specific definition.
- **Discrete I/O**: Discrete signals (ON/OFF) arise in the form of the Generic I/O or the Scan Chain. Thus, it is essential to have a way to extract that physical difference to allow identically processing them at higher layers. Discrete I/O provides that extraction thereby making all signals look like Generic I/O.

Generic I/O interfaces are parametrized which means the user can set the number of inputs, outputs and inouts that need to be grouped and shared to a specific Generic IO agent during the platform integration to the environment. Figure 3 clarifies details on the interface declaration that will eventually take the role of a device in the verification process.

Figure 4 demonstrates how several Generic I/O interfaces that are instantiated to support devices such as a power supply (psu1), the Southbridge, and several CPUs.

The fact that interfaces are parametrized may complicate agents in achieving a common or generic interface. One general solution to this resides in using classes to abstract parameterized interfaces, which is allowed by SystemVerilog's syntax. In this syntax, class objects representing interfaces are used at the driver and allow for monitoring so that the compile-time parameter is no longer needed. Specification of the parameter of the interface is voluntary for dynamic verification components and they can be coded independently for re-usability [3].

The Generic I/O agent has a standard UVM structure with its own sequencer, driver, and monitor. In addition, it possesses associative arrays accessed by strings. These arrays are the maps required for accessing the pinout in the interface. Each input, output, or inout will be registered in the agent by means of a configuration file and its name in the form of a string will be the access key. Figure 5 shows part of the configuration code for a set of inputs and outputs related to the Southbridge. After the Generic I/O agent is configured, the sequences must use these names to control and/or monitor the registered signals.

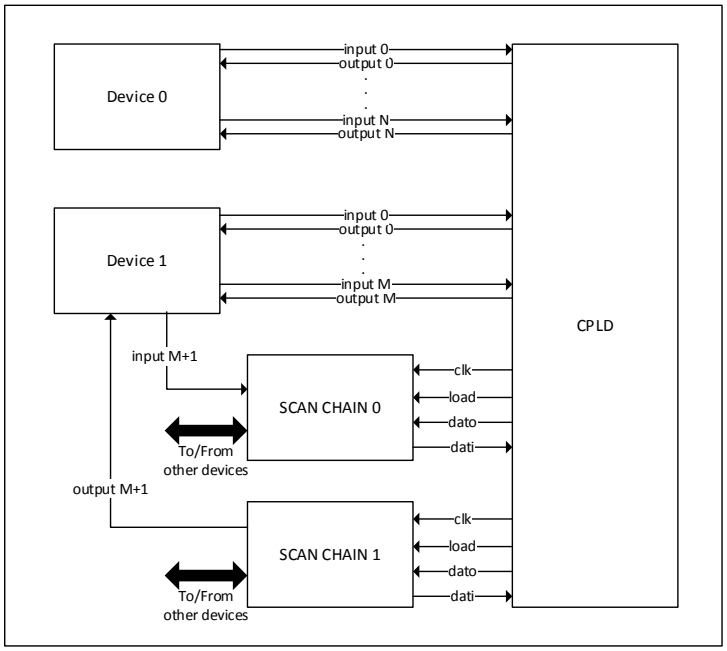

Figure 2: Illustrating CPLD Connecting to Other Server Devices

```
interface generic_io_intf#(int number_of_inputs = 1, //Number of intf's inputs.
                           int number_of_outputs = 1,//Number of intf's outputs.
                           int number_of_inouts = 1, //Number of intf's inouts.
                           string my_name = "generic_io_intf") //Intf's name.
                           ();

//Import abstract class package.
    import abstract_pkg::*;

//Interface's properties.
    logic [number_of_inputs -1 : 0] inputs;      //Inputs.
    wire  [number_of_outputs-1 : 0] outputs;     //Outputs.
    logic [number_of_inouts -1 : 0] fake_inouts; //Fake inouts (written by env).
    wire  [number_of_inouts -1 : 0] inouts;      //Real inouts.
```

Figure 3: Generic I/O Interface Definition

```
generic_io_intf #( 6,  3, 1) psu1_intf();
generic_io_intf #( 7, 11, 1) sbc_intf();
generic_io_intf #(14, 12, 1) cpu0_intf();
generic_io_intf #(14, 12, 1) cpu1_intf();
```

Figure 4: Generic I/O Interface Instantiations

```
input_names = '{"SBC_SLP_N",
                "PAL_SBC_CPUPWROK",
                "SBC_PLTRST_N"};

output_names = '{"PAL_SBC_PWROK_R",
                 "PAL_SBC_RSMRST_N",
                 "PAL_SBC_SYS_PWROK",
                 "PAL_SBC_THRMTRIP"};
```

Figure 5: Generic I/O Configuration Code for Registering SBC ON/OFF Signals

Another special aspect of the Generic I/O agent is found in its sequencer. It has a subscriber connected to the analysis port of the monitor in order to provide the sequences with a means to know the status of the inputs and outputs of the Generic I/O agent. The sequencer also accesses the maps configured in the agent. In this way it is possible to declare useful methods for reading and writing in a base sequence that facilitates the use of the Generic I/O package in upper layers. Figure 6 presents the declaration of the previously mentioned methods.

Closely related to the Generic I/O is the Discrete I/O agent which is merely an extension of the Generic I/O agent yet with additional support for inputs and outputs connected to the DUT via Scan Chain. The Discrete I/O's goal is to unify Generic I/O signals with those that belong to the same device but use Scan Chains to reach the CPLD. In this way all the sequences and methods can be reused despite platform-to-platform differences at physical level.

If a device is linked to a Discrete I/O agent during the integration of the platform, this design will be indicated through the configuration file of the environment which later will be responsible for making a factory override as illustrated in Figure 7.

At the checking level, the physical layer's only responsibility is to verify extremely specific cases such as the appearance of high impedances (z) or unknown values (x), thereby leaving a majority of the responsibility of the checking to the other two layers. This is because the physical layer is highly variable from one platform to another; thus, any specific code at this level requires an intensive time of development and maintenance. This increased developmental and maintenance cost, however, is the primary aspect to avoid while structuring the environment. For the same reason, the use of SVA was discarded as concurrent SVAs can only exist in the physical layer and would require having a special set of rules for each platform.

B. Device Specific Layer

The CPLD verification environment has a second layer called the Device Specific Layer which is populated with device specific sequences and checkers. Sequences in this layer model the behavior of the CPLD's surrounding devices in the server and each is specific to the type and version of the device (for the same device type and same version pinout). Accordingly, these sequences are selected and configured to the needs of the platform.

Device specific layer sequences appear in the form of initiators and responders. An initiator is a stimulus model that will initiate a transaction or multiple transactions with the DUT; in contrast, a responder reacts to outputs from the DUT and feeds stimulus back into the DUT [4]. To demonstrate, Figure 8 shows how a southbridge responder makes use of the methods provided by the Generic I/O package in order to implement SBC's runtime assertions logic. In this example, the responder waits for the CPLD to assert PAL_APWROK and PAL_PWROK prior to setting DRAM_PGD and CPU_PWROK. Then, it waits for PAL_SYS_PWROK to be asserted and finally de-asserts PLT_RST_N. This sequence of events can suddenly be truncated if a server shutdown request is detected.

In this device specific layer, all signals are treated as Generic I/O regardless of whether a signal is connected directly to the DUT or indirectly through a Scan Chain. Thus, at this point, the physical layer has been configured and signals can be accessed using device common names.

```
// UVM Macro for accesing the sequencer inside the sequence
    `uvm_declare_p_sequencer(generic_io_sequencer)

//User defined Methods:
    extern function bit find_signal(string signal_name, signal_dir dir,
                                    output int address);
    extern function int get_number_of_inputs();
    extern function int get_number_of_outputs();
    extern function int get_number_of_inouts();
    extern function void get_inputs_names(output string names[]);
    extern function logic read(int address, signal_dir dir = OUTPUT);
    extern function logic read_by_name(string signal_name,
                                       signal_dir dir = OUTPUT);
    extern task write(int address, logic write_value, signal_dir dir = INPUT);
    extern task write_by_name(string signal_name, logic write_value,
                              signal_dir dir = INPUT);
    extern task wait_change(int address, signal_dir dir = OUTPUT);
    extern task wait_change_by_name(string signal_name,
                                    signal_dir dir = OUTPUT);
    extern task wait_value(int address, logic value, signal_dir dir = OUTPUT);
    extern task wait_value_by_name(string signal_name, logic value,
                                   signal_dir dir = OUTPUT);
```

Figure 6: Generic I/O Declaration of Useful Methods

```
  foreach(registered_discrete_io_name[i]) begin
      string name = registered_discrete_io_name[i];
      set_inst_override_by_type(name,
                                generic_io_agent::get_type(),
                                discrete_io_agent::get_type());
      set_inst_override_by_type({name, ".", name, "_sequencer"},
                                generic_io_sequencer::get_type(),
                                discrete_io_sequencer::get_type());
      set_inst_override_by_type({name, ".", name, "_status_subscriber"},
                                generic_io_subscriber::get_type(),
                                discrete_io_subscriber::get_type());
  end
```

Figure 7: CPLD Environment Using Factory Placed Registered Discrete I/O Agents

```
fork:_sbc_runtime_asserts_block_
//Wait for both APWROK and PWROK to set or wait for sbc_off
//condition if SBC is forced to shutdown immediately.
    begin
        wait_value_by_name("PAL_APWROK", 1'b1, OUTPUT);
        wait_value_by_name("PAL_PWROK", 1'b1, OUTPUT);

        //Assert DRAMPWROK and PROCPWRGD.
        #60us;
        write_by_name("DRAM_PGD",  1'b1, INPUT);
        `uvm_info(get_name(),"Asserting DRAM_PGD with 1.", UVM_LOW)
        #60us;
        write_by_name("CPU_PWROK",  1'b1, INPUT);
        `uvm_info(get_name(),"Asserting CPU_PWROK with 1.", UVM_LOW)

        //Wait for SYS_PWROK, then de-assert PLTRST.
        wait_value_by_name("PAL_SYS_PWROK", 1'b1, OUTPUT);
        #150us;
        write_by_name("PLT_RST_N",  1'b1, INPUT);
        `uvm_info(get_name(),"Setting PLT_RST_N to 1.", UVM_LOW)
    end

    wait(sbc_off);
join_any
disable _sbc_runtime_asserts_block_;
```

Figure 8: SBC Responder Sample Code

The number and names of the Generic I/O agent instances depend on the platform as well as what devices are part of the specific server and are interacting with the CPLD. This information is readily specified in the environment's configuration object as shown in Figure 9. Device specific layer sequences can be set to run in a Generic I/O sequencer as a default sequence (this is the most common case for responders), in a virtual sequence, or in a test-case.

Checkers in the device specific layer must not only verify that the CPLD is interacting as expected with other devices but also is responsible for abstracting and sharing useful information for later processing in the functional layer. Each checker connects to the analysis port of a Generic I/O agent. It is in this way that it receives a callback when activity occurs in the interface, and afterwards, the component compares the received information against its expectations. It then updates the status and calculates new expectations. The media used by all checkers to share status, desired expectations, and other essential information to the functional layer is called the Analysis Bus. The Analysis Bus is an object in which several types of variables can be registered and are accessible by all checkers and scoreboards. Figure 10 demonstrates how a Voltage Regulator Module (VRM) checker assesses its information in relation to the Analysis Bus.

```
//Configure Generic I/O agents.
    m_env_cfg.add_generic_io_agent("cpu0");
    m_env_cfg.add_generic_io_agent("cpu1");
    m_env_cfg.add_generic_io_agent("pwrmisc");
    m_env_cfg.add_discrete_io_agent("riser1");
```

Figure 9: Generic I/O Agents Added to Environment by Means of a Configuration Object

```
//Register current power group.
    `ab_register_int_field("CURRENT_PG", current_power_group)

//Register General status.
    `ab_register_enum_field("GENERAL_STATUS", m_status, e_status)

//Register VRM Faults.
    `ab_register_enum_array($sformatf("%0s_FAULT", vrm[i].get_name()), m_fault, e_fault)

//Register each VRMs status: EN and PG ENUMS.
    `ab_register_enum_array($sformatf("%s_EN", vrm[i].get_name()), vrm, e_status, i, 1, en_s)
    `ab_register_enum_array($sformatf("%s_PG", vrm[i].get_name()), vrm, e_status, i, 1, pg_s)
```

Figure 10: VRM Checker Registering Several Variables in the Analysis Bus

C. Functional Layer

The Functional Layer is the third layer where functional sequences and functional scoreboards reside utilizing platform specific configuration objects in order to determine supported CPLD features and the devices connected to the DUT. The code in the Functional Layer is shared across platforms as the code performs identical functionalities in a specific server generation. In rare situations, special cases arise and may be patched with platform specific code by using the UVM factory to override sequences or components. On the stimuli side, regular UVM virtual sequences are revealed which are capable to control other device specific sequences in order to inject CPLD functional scenarios. At this point, test-cases may select, configure, and execute these virtual sequences.

In the Functional Layer, the checking part is quite remarkable as it is in this layer where functional scoreboards and coverage collectors reside. For that reason, these highly complex elements in the CPLD verification environment process and verify all the information registered in the Analysis Bus while at the same time are flexible enough for different configurations. Figure 11 exhibits the main attributes (or properties) of the scoreboard base class. These are complex objects each with a specific data processing logic.

```
class scoreboard_base extends checker_base;

// UVM Factory Registration Macro
    `uvm_component_utils(scoreboard_base)

//------------------------------------------
// Data Members
//------------------------------------------
    analysis_bus              abus;
    property_item             properties [string];
    state_machine_reference   state_machines [string];
    sequence_checker          sequences [string];
    the_maker                 maker;
    scoreboard_config         sb_config;
    coverage_collector        cc;
```

Figure 11: Scoreboard Base Declaration Attributes

Class-based languages are object-oriented and uphold two main categories – classes and instances. A class property_item is similar to an SVA property as it has the capability to check that Boolean conditions representing logical scenarios occur in certain timing circumstances. State_machine_reference is a class used as a behavioral model for most of the state machines present in the CPLD. In this class, it is possible to calculate expectations accordingly with the activity processed in each property_item.

As mentioned previously, the CPLD performs multiple sequencing tasks which are verified in the scoreboards using sequence_checker objects. Comparisons are performed between state machine reference models and the activity observed in the Analysis Bus. One coverage_collector instance is integrated in each functional scoreboard. Its primary task is to collect information about state machines transitions that reflect all the possibilities occurring in a specific CPLD functionality.

Finally, due to the complexity in managing and connecting all objects in order to interact and evaluate functional features, it was necessary to create a tool that provides users a simplified method to develop the scoreboards' logic. This tool is called "The Maker." A micro language designed for The Maker simplifies and accelerates scoreboard development. The functions provided by The Maker ask for a string variable as an argument. In this string, the user can specify the signals and conditions to evaluate by using the defined syntax, can identify the sequences to check, and can highlight the state machines to create and assess a myriad other features. The Maker will then read the string, analyze it, and proceed to create the objects and the logic to perform the specified function.

Figure 12 shows an example of how easy it is to evaluate Boolean conditions using The Maker. In this case, a property has been created for each of the ways that the server can use to start a power up sequence: ILOWAKEUP, PHYSICAL and VIRTUAL. For instance, the ILOWAKEUP property evaluates if the CPLD input signal GMT_WAKEUP_N is low. If it is, then the value of the property will be 1. Also, it shows that you can create properties conditionally which helps to reuse the same scoreboard for different

platforms. Later, these properties will be reused for creating the reference state machines and the sequence checkers that will complete the scoreboards.

In Figure 13 it is possible to observe the way a power sequencer scoreboard is defined using "The Maker" and its micro-language.

IV. Results

Over the last two ProLiant Server generations (G9 and G10), the HPE CPLD team has been using the methodology described in this document to successfully verify up to fifteen platforms per generation. G9 verification required an environment capable of supporting multiple platforms and the CPLD's non-transaction nature. This was possible by implementing Generic I/O components and sequences. The behavioral checking and coverage collection utilized concurrent SVAs for each platform. During the G10 verification, SVAs were substituted by functional scoreboards and coverage collectors in order to reuse checking logic and save maintenance time while also providing a more robust functional verification by adding more flexible components and development tools like "The Maker."

During G9 verification, the platform integration consumed up to 50% of the total verification time due to the high maintenance required by the SVA blocks and the fact that, at the physical layer, each platform had high variances resulting in a lower code reuse capability. After implementing class based functional scoreboards and coverage collectors in G10 generation, the integration time was reduced to approximately 10% to 15% of the verification effort.

```
//CPLD PAL_PWRBTN_N property
  maker.add_property ("PCH_BTN_PRSS",   "[PAL_PWRBTN_N_STATUS != 1]");

//Power up trigger related properties.
  if(sb_config.enable["has_ilowakeup"]) begin
      maker.add_property ("ILOWAKEUP", "[GMT_WAKEUP_N = 0]");
  end
  if(sb_config.enable["has_physical_btn"]) begin
      maker.add_property ("PHYSICAL", "[PWR_BTN_STATUS = POSEDGE]");
  end
  if(sb_config.enable["has_virtual_btn"]) begin
      maker.add_property ("VIRTUAL", "[RAL:vir_pwr_btn = 1]");
  end
```

Figure 12: Property declaration for ILOWAKEUP, PHYSICAL and VIRTUAL power up methods

```
//Power sequencer related properties.
    maker.add_property ("PSQR_RESET",    "[SM:POWER_SQR_REF_SM = RESET]");
    maker.add_property ("PSQR_STBY",     "[SM:POWER_SQR_REF_SM = STANDBY]");
    maker.add_property ("PSQR_S5VRDS",   "[SM:POWER_SQR_REF_SM = S5_ON_VRDS]");
    maker.add_property ("PSQR_S5_STDBY",{"[SM:POWER_SQR_REF_SM = S5_ON_VRDS] |",
                                         "[SM:POWER_SQR_REF_SM = STANDBY]"});

//Standby VRDs related properties.
    maker.add_property ("STBY_VRDS_OFF", "[AB:SBY:OVERALL_STATUS != ALL_ON]");
    maker.add_property ("STBY_VRDS_ON",  "[AB:SBY:OVERALL_STATUS = ALL_ON]");

//Chassis related properties.
    maker.add_property ("AMB_LED_OFF","[PWRBTN_AMB_STATUS = OFF]");
    maker.add_property ("AMB_LED_ON", "[PWRBTN_AMB_STATUS = ON]");

//CPLD RESET checking.
    maker.add_sm ("CPLD_RESET_SM",          //Name of the reference SM.
                 {"STBY_VRDS_ON  -> ON",    //ON state.
                  "!STBY_VRDS_ON -> OFF"}); //OFF state.

    maker.add_sc ("CPLD_RESET_CKR",     //Name of the sequence checker.
                 {""},                  //No transitions to check.
                 {"OFF -> PSQR_RESET",  //Expected functionalities
                  "OFF -> AMB_LED_OFF",
                  "ON  -> PSQR_STBY",
                  "ON  -> AMB_LED_ON"},
                  "CPLD_RESET_SM",      //Reference state machine.
                  "OFF",                //Initial state.
                  "ON");                //Final state.
    maker.add_reset ({"STBY_VRDS_OFF"}, "CPLD_RESET_CKR");
```

Figure 13: Power Sequencer Scoreboard's Logic Implemented Using "The Maker" for Reset Evaluations

It is anticipated, that future CPLD generations will be able to reuse up to 95% of the already developed verification code due to the fact that the CPLD keeps similar functionalities from one generation to another and due to the high configurability of the components such as the Generic I/O agent. The Generic I/O has proven been flexible enough to be used in CPLD block verification and even in other projects.

V. Conclusions

At HPE, for design engineers to meet the unique demands of our clients, it was necessary to create a digital system meeting both non-transaction based and multi-platform conditions. Per the analysis above, the design team was highly successful in developing a CPLD Verification environment capable of dealing with its non-transaction and multi-platform characteristic by using a layered approach with highly configurable components at the lower layers and highly reusable components at the top layer.

While SVA blocks are useful, implementing SVA blocks incurs a high cost maintenance in multi-platform verification environments due to the significant platform-to-platform differences at the physical level and also the fact that concurrent assertions can only be instantiated in modules and interfaces. The design team effectively created a tool for developing functional scoreboards and coverage collection logic. This tool has its own string based micro-language and provides the benefit of less complexity to checking code writers.

VI. References

[1] Hewlett Packard Enterprise Development LP, "HPE Integrated Lights-Out Portfolio," in Family data sheet 4AA4-5167ENW, February 2018, Rev. 6 [Online]. Available: https://h20195.www2.hpe.com/v2/getpdf.aspx/4aa4-5167enw.pdf

[2] B. Cohen, "SVA in a UVM Class-based Environment", [Online]. Available: https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-9_issue-1/articles/stream/sva-in-a-uvm-class-based-environment_vh-v9-i3.pdf

[3] W. Yun and S.Zhang, "Deploying Parameterized Interface with UVM" DVCon United States 2013.

[4] B.Wile, J.Goss and W.Roesner, "Comprehensive Functional Verification, The complete Industry Cycle", *Elsevier*, pp. 77-78, 2005.