Addressing Advanced Mixed-Signal Verification Scenarios by Developing a UVM Framework for Analog Models

Simul Barua
Ulkasemi Inc.
20045 Stevens Creek Blvd, Suite 2B
Cupertino, CA 95014
simul@ulkasemi.com

Shahriar Kabir
Ulkasemi Inc.
20045 Stevens Creek Blvd, Suite 2B
Cupertino, CA 95014
shahriar@ulkasemi.com

Henry Chang
Designer's Guide Consulting, Inc.
3043 Meridian Ave #35
San Jose, CA 95124
henry@designers-guide.com

***Abstract*- With the rise of AI, IoT, and automotive safety applications, the demand for reliable mixed-signal verification in Integrated Circuit (IC) design has become critical. Increasing design complexity, particularly at the interface of analog and digital components, often leads to verification gaps and costly respins. This paper introduces the Analog Mixed-Signal Verification (AMSV) utility framework, which enhances system-level testbenches by integrating analog behavioral models within a Universal Verification Methodology (UVM) environment. Leveraging SystemVerilog's object-oriented programming (OOP) capabilities, the AMSV framework enables precise monitoring and control of analog models, allowing for accurate simulation of real-world conditions, including temperature effects, parametric variations, and subsystem failures. Through detailed case studies, we demonstrate the framework's capacity to address verification needs in advanced applications, significantly improving accuracy and efficiency. Simulation results validate the framework's potential to enhance IC reliability, making it particularly relevant to emerging, high-stake domains.**

I. Introduction

The complexity of Integrated Circuit (IC) verification has significantly increased, often outpacing the complexity of the design itself. This trend is particularly evident in modern applications such as Artificial Intelligence (AI), the Internet of Things (IoT), and automotive safety, all of which demand heightened reliability and seamless functionality across both analog and digital domains. Consequently, verification success rates have declined, leading to a rise in costly IC respins [1]. Two significant factors driving these respins are issues in the analog design and bugs in the mixed-signal interfaces, which are crucial components in modern designs. Our previous research [2] addressed these challenges by incorporating basic analog functional models into chip-level verification environments, demonstrating how these models can enhance system-level verification to catch bugs that would otherwise evade detection in traditional digital-focused testbenches.

Despite these advancements, conventional verification methodologies still lack the robustness needed to address more complex system requirements. As IC design complexity evolves, we must account for system performance, safety, and other more complex scenarios that a digital twin, which needs to include analog elements, must address. This could involve simulating the effects of temperature or parametric variations of block non-idealities onto the overall system for performance. For safety, this might mean modeling how analog blocks, subsystems, and systems fail in particular ways.

This is where analog models demonstrate their true value. Besides simulating at feasible speeds compared to transistor-level simulations, models can incorporate complex effects like environmental stress factors and parametric shifts in a straightforward manner. The potential of these analog models is vast, offering greater flexibility, enabling the modeling and simulation of complex real-world scenarios, and allowing broader system assessments, such as evaluating how AI algorithms perform when critical analog components degrade or fail.

However, to fully harness these enhanced analog models, the traditional system-level testbench widely developed in SystemVerilog and/or Universal Verification Methodology (UVM) must also evolve. They require improved capabilities for monitoring, controlling, and interacting with analog models, enabling verification under advanced, real-world conditions.

This paper introduces the Analog Mixed-Signal Verification (AMSV) utility framework, which leverages SystemVerilog object-oriented programming (OOP) constructs to enhance analog models and the system-level testbench. The AMSV utility framework goes beyond simply addressing respin issues; it enables precise modeling and simulation of real-world conditions such as temperature effects, parametric variations, failure conditions, analog impairments, and other intricate scenarios. Thus, it allows for a more comprehensive evaluation of system performance and safety.

II. Motivation and Purpose of the AMSV Utility Framework

Traditionally, simulating environmental effects required slow, transistor-level simulations, which provided accuracy but were impractical for system-level verification and offered no direct control over environmental impacts from testbenches. This limitation made it challenging to verify critical features, such as the system responses to temperature-induced frequency drift of a Phase Locked Loop (PLL) or the robustness of safety mechanisms like thermal shutdowns. Moreover, verifying safety conditions at the transistor level is notoriously challenging and designing system-level tests to intentionally induce predictable failures is only possible if the design inherently accommodates fault conditions. These limitations highlight the need for a more flexible and efficient verification approach that could seamlessly handle complex, mixed-signal scenarios.

The AMSV utility framework we will present addresses these challenges by enabling analog models to incorporate real-world behaviors. Additionally, the framework empowers the UVM testbench to dynamically set and adjust parameters, offering real-time control and monitoring of environmental conditions and fault scenarios, effectively verifying these features and ensuring that system responses to changes, such as frequency adjustments or safety reactions, are accurate and reliable.

The AMSV framework also facilitates continuous fault detection by enabling block-level analog models to report faults directly to the UVM testbench. This capability enhances fault detection and debugging, allowing verification teams to comprehensively test critical behaviors, from environmental resilience to safety mechanisms, and significantly advance full-chip verification for today's complex systems.

Essentially, what is required in addition to the input/output ports of the blocks in the system is a set of potentially bidirectional ports into the model of the block (not the schematic) that can control the model or from which to make observations for verification purposes. These additional ports cannot be added to the port list of the model as that would invalidate the pin accuracy of the model, which is required for model vs. schematic (MVS) checking [3] and for netlisting. Therefore, a separate mechanism must be introduced for these ports. This is where the AMSV framework plays a role.

In the following sections, we will explore the details of the framework and demonstrate how it enhances both analog models and the system-level testbench, ultimately improving the system-level verification of today's complex mixed-signal designs.

III. Structure of the AMSV Utility Framework

The AMSV utility framework is structured as a SystemVerilog package that includes OOP classes, lookup tables (LUTs), pre-defined macros, and other specialized utilities. This modular design integrates seamlessly with analog models and the UVM testbench, allowing for dynamic parameter control, real-time fault injection, and efficient communication between the analog models and the verification environment. Fig. 1 illustrates the framework's architecture, highlighting the core components: the model and global parameter LUTs, along with the utility methods to help with the communication and management of the components.

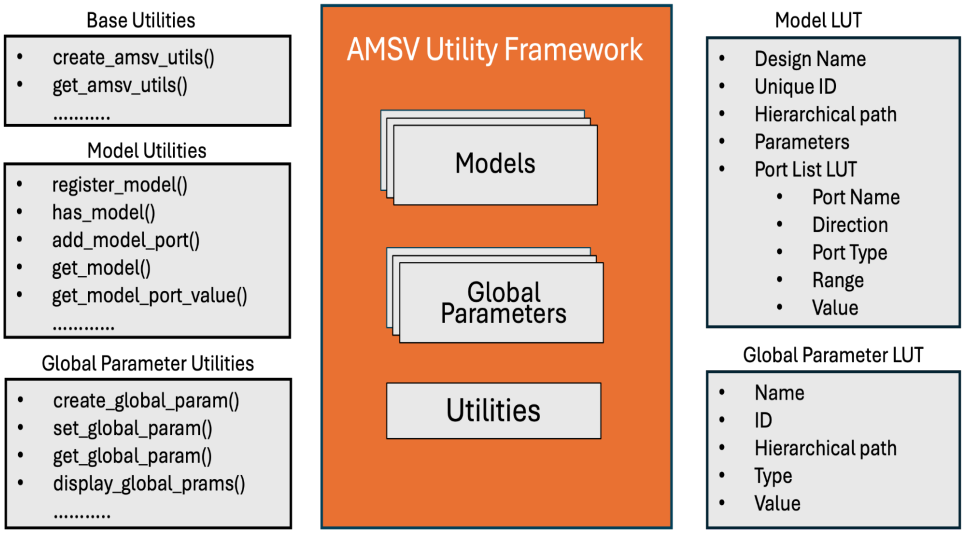

Figure 1. Visual illustration of the structure of the AMSV utility framework.

A. Model LUT

The model LUT contains a comprehensive set of configuration settings and parameters specific to each analog model. This table includes essential information, such as a list of models, port details (including direction, type, and additional settings), and configuration parameters. Moreover, the model LUT facilitates access to the framework's global variables across models, streamlining parameter management and ensuring consistency in system-wide settings.

This structure enables analog models to report input range faults directly to the UVM testbench, allowing for the rapid detection of deviations from expected operating conditions. With capabilities to monitor block-level pin values and track critical metrics, the model LUT is instrumental in real-time model adjustments and fault reporting, making it essential for verifying complex mixed-signal behaviors.

B. Global Parameters LUT

The global parameter LUT serves as a centralized resource for storing parameters that affect multiple models or the entire verification environment. It acts as a unified source for environmental variables and system-wide settings. Typical entries include temperature, reference voltages, block failure mechanisms, and other key parameters.

The global parameter LUT allows seamless access and real-time updates from the testbench, enabling the UVM environment to dynamically influence model behavior as needed. Accessible to both the testbench and analog models, it ensures consistent parameter application across the verification environment. This structure guarantees that critical conditions are uniformly maintained, facilitating the reliable verification of complex mixed-signal interactions.

C. Utility Methods

The AMSV framework includes a suite of utility methods that establish the primary communication channels between the UVM testbench and the analog models. These methods provide essential tools for model management, global parameter control, and signal monitoring, making the framework highly adaptable to various verification scenarios.

- Base Utilities: These methods manage and integrate the AMSV framework itself. They facilitate the creation of singleton class objects within the framework, ensuring a seamless setup and preventing integration errors, which helps maintain consistency and reliability in the use of AMSV utilities.
- Model Utilities: Designed for model management, these utilities register each analog model within the framework, storing relevant information in the model LUT. This process allows the UVM testbench to monitor and handle various scenarios by accessing the model data as needed.
- Global Parameter Utilities: These functions enable the testbench to dynamically set global parameters and retrieve real-time values from the models. By providing centralized control over parameters, they allow for real-time adjustments to model behavior, which is essential for simulating complex scenarios.

D. Code Snippet

Fig. 2 provides an overview of the structure of the AMSV utility framework, illustrating how the main class organizes core components, including lookup tables and utility functions.

The AMSV package is designed to encapsulate the model and global parameter LUTs while providing utility functions that streamline communication between models and the testbench.

```
package amsv_utils_pkg;
  // Model LUT
  class model_utilities;
  // Global Paremeter LUT
  class global_utilities;
  // Other Helper Signals of the package
  ........................
  // The Base Class of the AMSV Utility Framework
  class amsv_utilities;
    // AMSV utility singleton class
    static amsv_utilities amsv_utils = null;
    // Lookup table consisting model details
    model_utilities models[string];
    // Lookup table for global variables
    global_utilities globals[string];
    ........................
    // Utilities
    // Private constructor to prevent direct instantiation
    extern protected function new(string name);
    // Create instance of AMSV Utility
    extern static function amsv_utilities create_amsv_utils(string name);
    // Get instance of AMSV Utility
    extern static function amsv_utilities get_amsv_utils();
    ........................
    // Model LUT Utilities
    // Registers Analog model in lookup table with name, unique ID and hierarachy
    extern function void register_model(string name = "", string id = "", string hier = "");
    // Returns 1 when model exists with provided unique ID
    extern function bit has_model(string id);
    // Adds port information of registered model
    extern function void add_model_port(string id, string name, string dir, string port_type);
    // Returns model utility based on unique ID
    extern function model_utilities get_model(string id);
    // Returns the port description of the model based on unique model ID and port name
    extern function port_t get_model_port(string id, string port_name);
    ........................
    // Global Parameter Utilities
    // Creates global variable with name, data type and value
    extern function void create_global_param(string name = "", string data_type = "", real value);
    // Sets global value during runtime
    extern function void set_global_param(string name, real value);
    // Returns global value with provided name
    extern function real get_global_param(string name);
    // Returns 1 when global value with provided name exists
    extern function bit has_global_param(string name);
    ........................
    // Display Utilities
    ........................
  endclass
endpackage
```

Figure 2: AMSV utility framework package.

E. Pre-defined Utility Macros

The AMSV framework also includes a set of pre-defined utility macros designed to simplify integration and manage various operations. These macros minimize the need to type repetitive code, ensure consistent usage across analog models and verification environments, and reduce code complexity, making the framework more accessible and efficient.

Key macros provided by the AMSV framework include:

- Model registration macros: These macros streamline the setup of each analog model by initializing model-specific configurations within the model LUT. This includes pin mappings and parametric settings, ensuring that each model is correctly registered and accessible to the UVM testbench, thereby reducing the setup time for individual models.
- Global parameter synchronization macros: These macros enable continuous observation of global parameters set by the testbench, allowing models to react to parameter changes and adjust their behavior accordingly.
- Signal monitoring macros: These macros facilitate the monitoring of critical signals within analog models, enabling the UVM testbench to detect and report deviations from expected behavior, which streamlines fault detection and debugging.
- Fault reporting macros: These macros simplify block-level fault detection by enabling direct fault reporting from the models to the testbench, allowing the UVM environment to manage fault conditions efficiently.

Fig. 3 shows examples of definitions of some key macros in the AMSV framework. While only the definitions are shown, they illustrate how these macros support streamlined operations within the framework.

```
`amsv_utils_register_model(model_name, unique_id, pin_list, param_list)
// Registers the model with specified pins and parameters in the Model LUT.

` amsv_utils_sync_globals(VAR_NAME, GLOBAL_PARAM_NAME)
// Synchronizes global parameter: GLOBAL_PARAM_NAME value set by the testbench
// for continuous observation by the models.

`amsv_utils_monitor_signal(model_name, hierarchical_path, signal)
// Enables real-time monitoring of a specified signal, allowing the
// testbench to track and debug signals.

`amsv_utils_report_fault(fault_condition, message)
// Reports a fault condition directly from the model to the UVM testbench
```

Figure 3: Key macro definitions for simplified integration of the AMSV utility framework.

These macro definitions illustrate how the AMSV framework simplifies tasks such as model registration, global parameter synchronization, signal monitoring, and fault reporting. By utilizing these macros, both modeling and verification teams can quickly configure the AMSV framework and focus on verifying complex behaviors that traditional setups, reliant on hierarchical references or fixed testbench names, often struggle to address. This saves time and enables a more efficient and reliable verification process.

In the next section, we will examine case studies that showcase the AMSV framework in real-world verification scenarios, illustrating how its components enhance the accuracy and efficiency of system-level mixed-signal verification.

IV. Case Study: Application of AMSV Utility Framework to Mixed-Signal System Design

In this section, we demonstrate the application of the AMSV framework using an example System-on-Chip (SoC) design comprising both digital and analog components. Our primary objective is to showcase how to integrate the framework with analog models and the system-level UVM testbench and how this integration enables the verification of complex scenarios.

A. Overview of the Design and Verification Environment

The example design includes critical mixed-signal blocks, specifically a power management unit and a phase-locked loop (PLL). The AMSV framework is employed to model the impact of environmental factors, particularly temperature, on these analog components, allowing for a more thorough verification of system-level responses under varying conditions.

The PLL block in this design is sensitive to temperature fluctuations, which causes its operating frequency to drift from the intended value. This frequency deviation is monitored by a control unit within the digital subsystem, which dynamically adjusts the PLL to maintain its desired frequency. Additionally, the design incorporates a temperature monitoring and thermal shutdown (TSD) block that is responsible for overseeing the system temperature. If the temperature exceeds the operational threshold, the TSD block is designed to shut down the system to prevent overheating and catastrophic failure.

Using the AMSV framework, we model the temperature effect on the PLL to verify whether the control unit appropriately compensates for frequency drifts caused by rising temperatures. Furthermore, to assess the system's response to safety mechanisms, we include a failure mechanism in the thermal shutdown block using the framework. This simulated fault prevents the TSD block from triggering, even when temperatures rise above safe limits. This allows us to analyze how the system reacts under potential safety-compromising conditions.

The AMSV framework facilitates the verification of these complex scenarios by integrating them into the UVM testbench. We will describe how to configure parameters such as temperature and trigger the TSD failure mechanism directly from the UVM testbench. We will then present specific test cases developed to validate these safety and performance features. Fig. 4 provides a block diagram of the verification setup, illustrating the design under test (which includes digital blocks and analog models) and highlighting the role of the AMSV utility framework as an interface between the models and the testbench. The following sections will provide detailed discussions of how the AMSV framework is incorporated into both the analog models and the UVM testbench, as well as the specific test cases developed to validate these complex scenarios.

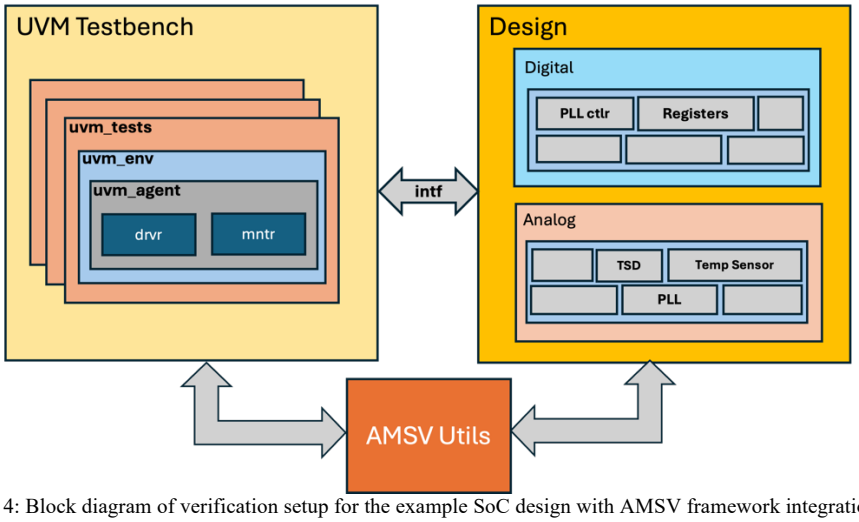

Figure 4: Block diagram of verification setup for the example SoC design with AMSV framework integration.

B. Modeling Analog Blocks and Integrating the AMSV Framework

We developed fully functional models for the analog subsystem using the SystemVerilog discrete electrical (also known as the user-defined nettype) approach. These models were generated with the Model-in-Minutes (MiM) tool [4], which produces distinct analog mixed-signal (AMS) and discrete mixed signal (DMS) models in Verilog-AMS and SystemVerilog respectively, and self-checking model-versus-schematic (MVS) testbenches for analog blocks from a specification. MiM allows for incorporating custom user code into model containers (shared settings that all models inherit within a project). This feature was particularly beneficial for integrating the AMSV utility framework into multiple models without modifying each specification independently.

Fig. 5 illustrates the MiM specification used to construct the Voltage Controlled Oscillator (VCO) of the PLL, specifically modeling how the system temperature impacts its behavior. By utilizing the user code section of MiM, we integrated the AMSV framework by embedding the necessary macros directly into the generated models.

Specifications Name: **VCO**

Description: **PLL Voltage Controlled Oscillator with AMSV Utility Framework**

Ports (Hide)

Name	Dir	Port type	Description	Range	Behavior
out	output	digital	VCO Output		osc
in	input	voltage	Input control voltage of the VCO		
[7:0]vcoCF	input	digital	Settings for VCO center frequency	nox	vcoCF_val with protect
enable	input	digital	Enable Pin		On = enable && !Fault with smooth
bias	input	ibias	Input Bias Current	5uA to 20uA	
vdda	input	supply	Supply	1.7V to 1.9V	I = On*10u
gnda	input	ground	Ground	-10mV to 10mV	

Discrete Variables (Hide)

Name	Var type	Description	Value	Trigger	Enable	Initial	Units
temp	real	Local variable to sync with AMSV Global Temperatur Parameter				27	C
Kvco	wreal	Voltage sensitivity	25M				MHz/V
delta_T	wreal	Temperature Difference	(temp+273)-TEMP_NOMINAL				K
temp_freq_drift	wreal	Frequency drift due to temperature	TEMP_COEFF*delta_T				Hz
Ctrl	real	Control Voltage	clip(V(in) - 1.25, -500m, 500m)	posedge osc			V
F0	wreal	Center Frequency	2.374G + 500k*vcoCF_val				Hz
f	wreal	Oscillator Frequency	F0 + (Kvco*Ctrl)-temp_freq_drift				Hz
osc	reg	Oscillator	#(0.5/f) ~osc		enable	0	

User Code (Hide)

Location	Code to Insert

DE model: before module	`// Importing AMSV Utility Framework Package`
	`import amsv_utils_pkg::*;`
	`// Include the AMSV Utility Macros`
	`` `include "amsv_utils_macros.sv" ``

DE model: after port declarations	`// Registering model to AMSV Utilities` `` `amsv_utils_register_model("VCO", "vco") ``
DE model: after variable declarations	`// Sync Global Variable 'temperature'` `` `amsv_utils_sync_globals(temp, "temperature") ``

Figure 5: MiM specification of the voltage-controlled oscillator (VCO) of the PLL with AMSV utility framework integration.

Fig. 6 shows the model created from the MiM specification, emphasizing the inclusion of AMSV utility macros. The *`amsv_utils_register_model* macro registers the model within the framework's Model LUT by providing the design name and a unique identifier. To synchronize the model with the global temperature parameter stored in the Global LUT, we utilized the *`amsv_utils_sync_global* macro, assigning the internal variable *temp* to store the global temperature value.

The code also demonstrates block-level assertions generated by MiM based on the range column in the ports table shown in Fig. 5. These assertions detect and report block-level faults when input values deviate from normal operating conditions. Leveraging the AMSV utility framework, these crucial block-level assertions can be reported directly to the system-level testbench, facilitating debugging and early bug detection in the design.

To test safety-critical scenarios, we integrated a failure mechanism into the thermal shutdown (TSD) model using the AMSV utility framework. This framework synchronizes with a global parameter called *tsd_force_fail*, utilizing the *`amsv_utils_sync_globals* macro. By connecting the global parameter *tsd_force_fail* with the TSD model, we can override the default behavior of the TSD directly from the testbench. When *tsd_force_fail* is enabled, the TSD bypasses its normal shutdown response, even as the temperature rises. Fig. 7 illustrates the code implementing this failure mechanism within the TSD model.

```
`timescale 1s/1ps

// Importing AMSV Utility Framework Package
import amsv_utils_pkg::*;
// Include the AMSV Utility Macros
`include "amsv_utils_macros.sv"
// MODULE HEADER  {{{1
module VCO (
output out,                          // VCO Output
input in,                            // Input control voltage of the VCO
input[7:0] vcoCF,                    // Settings for VCO center frequency
input enable,                        // Enable Pin
input bias,                          // Bias Input
input vdda,                          // Supply
input gnda                           // Ground
);
// Registering model to AMSV Utilities
`amsv_utils_register_model("VCO", "vco")

// Discrete-Electrical (DE) Transceivers  {{{2
real in$Vobs, in$Iobs, in$Idrv, in$Gdrv;
DE_norton Xtcvr_in (in, in$Vobs, in$Iobs, in$Idrv, in$Gdrv);
.........................

// Local variable to sync with AMSV Global Temperature Parameter [C]
real temp = 27;
.........................
// Sync temp variable with Global Parameter 'temperature'
`amsv_utils_sync_globals(temp, "temperature")
.........................
// Discrete Variables  {{{2
assign Kvco = 25e6;
assign delta_T = (temp + 273) - TEMP_NOMINAL;
assign temp_freq_drift = TEMP_COEFF*delta_T;
.........................
always @(posedge osc)
Ctrl <= min(max(in$Vobs - 1.25+0, -500e-3+0), 500e-3+0);
assign F0 = 2.374e9 + 500e3*vcoCF_val;
assign f = F0 + (Kvco*Ctrl) - temp_freq_drift;
always wait (On)
osc = #(`fromSeconds(max(500e-3/f, `TIME_PREC))) ~osc;
// Port Drivers  {{{2
assign out = osc;
.........................
// ASSERTIONS  {{{1
// Check for V(vdda) out of range (1.7V to 1.9V)  {{{2
assign #(`fromSeconds(25e-9), 0) _faults.V_vdda = !((
(1.7 <= vdda$Vobs) && (vdda$Vobs <= 1.9))
|| !_locals.enableAsserts);
always @(_faults.V_vdda)
_reportFault(_faults.V_vdda, "V(vdda) out of range");
assign _FaultOnInput = _faults.V_vdda;
.........................

endmodule
```

Figure 6: AMSV utility framework-enhanced VCO model with global temperature parameter effect.

```
`timescale 1s/1ps
// Importing AMSV Utility Package and Macros
import amsv_utils_pkg::*;
`include "amsv_macros.sv"
// Module Declaration
module TSD(
    tsd,
    trim,
    enable,
    por_b,
    i_ptat,
    vdda,
    gnda
);
// Parameters
.....................
// Port Types and Declaration
.....................
// Registering model to AMSV Utilities
`amsv_utils_register("TSD", "PLL-TSD")

// Syncing with Global Variable tsd_force_fail
`amsv_utils_sync_globals(tsd_force_fail, "tsd_force_fail")

// Discreate Behavior
// Calculate temperature from i_ptat
.....................

// Drive TSD
// If tsd_force_fail = 1 tie the output to 0
assign tsd = (tsd_force_fail==0) ? tsd_reg : 0;

// Analog Assertions
.....................
endmodule
```

Figure 7: Code snippet of TSD model synchronization with global parameter *tsd_force_fail* to simulate failure mechanism.

C. *Integrating the AMSV Framework to UVM System Level Testbench*

To enable the AMSV utilities within the UVM testbench, we start by importing the *amsv_utils_pkg* into the testbench top level and creating a singleton instance of the AMSV utility class using the *create_amsv_utils* method provided in the package. This singleton instance can then be accessed throughout the UVM testbench by setting it via the UVM configuration database. With this setup, the UVM testbench can fully leverage AMSV utility methods, enabling the generation of real-world scenarios that were previously challenging to recreate with traditional approaches. Fig. 8 illustrates the code used to integrate the framework into the top level of the UVM testbench and configure it via the UVM configuration database.

```
import uvm_pkg::*;
//importing amsv_utils package and macros
import amsv_utils_pkg::*;
`include "amsv_utils_macros.sv"

module tb_top_ahb;
  // PLL DUT instance
  pll_top DUT_top (...);

  initial begin
    // Creating amsv singleton instance
    amsv_utilities amsv_utils = amsv_utilities::create_amsv_utils("top");
    // Set the amsv instance in the uvm testbench environment
    uvm_config_db#(amsv_utilities)::set(null, "*", "amsv_utils", amsv_utils);
    ...
    run_test();
  end
endmodule
```

Figure 8: AMSV utility framework in UVM testbench top-level with configuration database integration.

Once the AMSV utility object is accessible in the test case (through the configuration database inside the *build_phase*), it can be used to initialize parameters that impact the model's behavior. In the code shown in Fig. 9, we initialize the global parameter called *temperature* and define a variable called *tsd_force_fail* to simulate failure in the TSD block under critical conditions, allowing us to verify system responses to extreme scenarios.

```
class pll_test extends uvm_test;
    `uvm_component_utils(pll_test)
    ..................
    virtual function void build_phase( uvm_phase phase);
        .......
        // Getting AMSV Utility from tb_top
        if(!uvm_config_db #(amsv_utilities)::get(this,"","amsv_utils",amsv_utils)) begin
            `uvm_fatal("AMSV utilites","AMSV utility is not found")
        end
        ..................
    endfunction
    virtual function void connect_phase( uvm_phase phase);
        ..................
        // Creating global parameters
        amsv_utils.create_global_param("temperature", "real", 27.0);
        amsv_utils.create_global_param("tsd_force_fail", "bit", 0);
        ..................
    endfunction
    ..................
    task run_phase(uvm_phase phase);
        ..................
    endtask
endclass
```

Figure 9: Initialization of AMSV utility global parameters in UVM test case

The following section explores several key verification scenarios made possible with the AMSV Utility framework, utilizing the initialized parameters shown in Fig. 9.

D. Test Cases for AMSV Utility Global Parameter Variation and System Behavior Verification

We subjected the PLL to several test cases to thoroughly evaluate it under various conditions set by the AMSV utility framework.

1) The global temperature is initialized to a nominal value of 27°C, where we expect the PLL to achieve a stable lock.

2) The temperature is gradually increased to 60°C, and we anticipate the PLL maintaining its lock, as the feedback loop can compensate for this slow temperature change.

3) Next, the temperature is rapidly increased to 65°C. We expect the PLL controller from the digital side to adjust the central frequency of the VCO and achieve the PLL lock within a 250µs limit, verifying the system's ability to handle sudden thermal changes.

4) The temperature is set beyond the operational range, reaching 85°C, where the TSD block is expected to trigger and shut down the PLL for safety. This test verifies the TSD's protective functionality.

5) The temperature is then lowered to 65°C, within the safe range. We expect the PLL block to resume operation with the TSD block becoming inactive, ensuring proper recovery from shutdown.

6) Using the AMSV Utility, we simulate a failure in the TSD block by setting the parameter *tsd_force_fail* to 1, disabling the TSD's shutdown function. In this scenario, we set the temperature to 90°C to observe whether the system can recover the PLL and whether it has any internal safety features to ensure reliable functionality under such critical conditions.

Fig. 10 illustrates the implementation of these test cases within the *run_phase* of the UVM test case code, highlighting how the global parameters of the framework are adjusted for each scenario to assess the system's response to varying conditions.

```
class pll_test extends uvm_test;
    ..................
    task run_phase(uvm_phase phase);
        super.run_phase(phase);
        phase.raise_objection(this);
            // Starting the simulation with nominal temperature
            amsv_utils.set_global_value("temperature", 27)
            // Wait for the power-up sequence to finish
            ..................
            // Check if the PLL lock interrupt is high within 250us
            check_pll_lock_time(250e-6);
            // Vary the temperature to 60C and check the PLL is still locked
            for (real r=27; r <= 60; r=r+1 ) begin
                amsv_utils.set_global_value("temperature", r);
                #1us;
                check_pll_lock_time(0);
            end
            #250us;
            // Setting temperature to 65C and check if the system adjust
            // the VCOCF control to get the desired frequency
            amsv_utils.set_global_value("temperature", 65);
            check_vcocf_register(.temp(65));
            check_pll_lock_time(250e-6);
            // Setting temperature to 85 C and check if the TSD is activated
            amsv_utils.set_global_value("temperature", 85);
            check_tsd(1);
            ..................
            // Check if TSD is disabled once Temp is lowered to 65C
            // and if PLL becomes locked within 250us
            amsv_utils.set_global_value("temperature", 65);
            check_tsd(0);
            check_pll_lock_time(250e-6);
            ..................
            // Forcing TSD to fail through global params
            // and setting temperature to 90C and check if
            // the design has any safety mechanisms for TSD failure
            amsv_utils.set_global_value("tsd_force_fail", 1);
            #1us;
            amsv_utils.set_global_value("temperature", 90);
            check_tsd(0);
            check_vcocf_register(.temp(90));
            check_pll_lock_time(250e-6);
            ..................
         phase.drop_objection(this);
    endtask
endclass
```

Figure 10: UVM test case for system behavior verification with AMSV utility global parameter changes

V. Results & Discussion

Based on the test case scenarios described above, we can observe the system's response to temperature changes and safety conditions enabled by the AMSV Utility framework. Fig. 11 presents the simulation waveform, annotated to illustrate results corresponding to test cases 1 to 6. Each annotation marks a distinct scenario, highlighting key observations such as PLL lock status, temperature changes, TSD activation, and system recovery, providing a visual representation of the system's response under each test condition.

The following results provide insights into how the design behaves under each test case scenario:

1) The temperature parameter is initially set to a nominal value of 27°C, and the power-up sequence is initiated, resetting the design using the *porb* signal. After the reset sequence completes, the PLL establishes a lock, indicated by the lock bit being set to high on the waveform. This initial setup confirms the system can achieve stability under normal temperature conditions.

2) When the temperature is gradually raised to 60°C, the PLL maintains its lock as expected, demonstrating that the loop compensates effectively for a slow-moving temperature increase. This gradual adjustment simulates typical operating conditions where the system must adapt to minor thermal fluctuations without losing stability.

3) With an abrupt temperature rise to 65°C, the PLL lock disengages, and the system attempts to adjust the VCO center frequency, denoted as *vcoCF* on the waveform, to reestablish the lock. We observe that the PLL lock is successfully re-engaged once the *vcoCF* register sets an appropriate center frequency. This scenario verifies the system's ability to react to sudden thermal shifts, adjusting internal parameters accordingly to maintain functionality.

4) When the temperature is set to 85°C, beyond the operating range of 27°C to 80°C, the thermal shutdown (TSD) mechanism activates. At this point, the *tsd* signal is set to high, disabling the *pll_clk* and *sys_clk* signals to protect the system. This test validates the TSD mechanism's ability to prevent operation in unsafe thermal conditions.

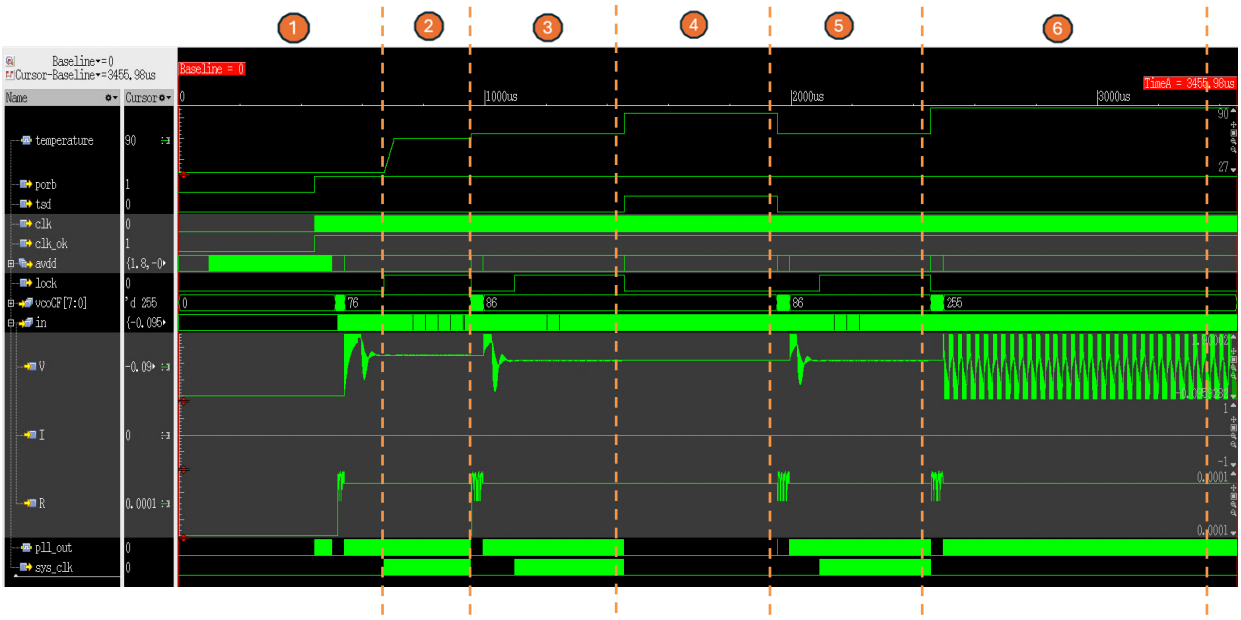

Figure 11: Simulation Results Waveform Illustrating System Responses to Temperature Variations and TSD Scenarios

5) The temperature is then lowered to 65°C, within the operational range. As expected, the *tsd* signal resets to low, and both *pll_clk* and *sys_clk* signals are reactivated, allowing the PLL to resume operation. This behavior confirms that the system can recover correctly and safely from a thermal shutdown once conditions are stable.

6) To test system resilience under a TSD failure, the *tsd_force_fail* parameter is set, and the temperature is raised to 90°C. The *tsd* signal does not activate in this scenario despite exceeding safe limits. The system attempts to compensate by setting the *vcoCF* register to its maximum value ('d255) to regain the PLL lock; however, it is unsuccessful, resulting in the PLL frequency fluctuation as it continuously tries to match the phase. By simulating this failure scenario, the designer gains insight into potential vulnerabilities and behavior under TSD malfunctions, enhancing system reliability through real-world scenario testing.

Overall, these observations demonstrate the AMSV utility framework's capability to simulate diverse environmental conditions and fault scenarios, providing invaluable insights into system behavior and robustness. The entire simulation was completed in approximately two minutes—a significant improvement over traditional transistor-level simulations, which cannot achieve such performance for comparable conditions. Furthermore, conventional testbenches often lack control over dynamic scenarios like abrupt temperature changes and fault conditions, making it challenging to validate complex interactions and safety mechanisms in real time.

IV. Conclusion

This paper presented the AMSV utility framework as an innovative approach to address mixed-signal verification challenges in modern IC designs. By integrating analog models with the UVM testbench, the framework enables efficient system-level verification previously unattainable with traditional methods.

Through the test cases, we demonstrated the framework's capability to simulate complex real-world scenarios, such as temperature fluctuations, abrupt environmental changes, and fault conditions, all within a short simulation time of approximately two minutes. These capabilities highlight the AMSV utility framework's ability to overcome the limitations of transistor-level simulations and conventional testbenches, which lack the flexibility and speed to handle such intricate conditions.

Beyond the scenarios presented, the AMSV utility framework offers other valuable applications. It provides a robust solution for verifying mixed-signal connectivity, ensuring reliable interactions between analog and digital components. With real-time monitoring and direct access to block-level signals, verification teams can develop system-level assertions and enhance coverage of analog designs. This capacity for connectivity verification, assertion development, and detailed coverage metrics makes the AMSV framework a versatile tool that goes beyond traditional verification setups.

The AMSV utility framework bridges crucial gaps by facilitating direct parameter control and real-time observation, reducing limitations that have long hindered traditional setups. Its ability to replicate complex environmental scenarios and dynamically interact with system parameters sets a new standard in efficiency, accuracy, and completeness for

mixed-signal verification. As IC designs continue to grow in complexity, the framework provides a scalable and adaptable solution that can keep pace with evolving verification needs, making it a valuable asset for developing advanced applications in areas such as automotive safety, IoT, and AI-driven systems.

In conclusion, the AMSV Utility framework represents a significant advancement in the field of mixed-signal verification, equipping engineers with the tools necessary to meet today's verification demands and setting the stage for future innovations. Future extensions of this framework may include expanded model libraries and enhanced utility functions, further strengthening its role in the verification process.

REFERENCES

[1] B. Bailey, "Trouble Ahead for IC Verification," Semiconductor Engineering, May 29, 2024. Available: https://semiengineering.com/trouble-ahead-for-ic-verification/. [Accessed: Sep. 08, 2024].

[2] S. Barua, F. Farshad, and H. Chang, "Advanced UVM Based Chip Verification Methodologies with Full Analog Functionality," DVCON USA 2024. Available: https://dvcon-proceedings.org/document/advanced-uvm-based-chip-verification-methodologies-with-full-analog-functionality/. [Accessed: Sep. 08, 2024].

[3] Kundert, Kenneth, "Model Validation (MVS)" from "Filling the Gap Between Digital and Analog Verification, Part 2", October 24, 2024, Available: https://designers-guide.com/main/webinars/?1=331#vid1. [Accessed: Nov. 01, 2024].

[4] Designer's Guide Consulting, Inc., "Analog Verification Products: Models in Minutes," [Online]. Available: https://designers-guide.com/main/products/. [Accessed: Nov. 01, 2024].