2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Reuse of System-Level Verification Components within Chip-Level UVM Environments

Diego Alagna, STMicroelectronics, Milan, Italy (*diego.alagna@st.com*)
Marzia Annovazzi, STMicroelectronics, Milan, Italy (*marzia.annovazzi@st.com*)
Alessandro Cannone, STMicroelectronics, Milan, Italy (*alessandro.cannone1@st.com*)
Marcello Raimondi, STMicroelectronics, Milan, Italy (*marcello.raimondi@st.com*)
Simone Saracino, STMicroelectronics, Milan, Italy (*simone.saracino@st.com*)
Mukesh Chugh, MathWorks, Natick MA, USA (*mchugh@mathworks.com*)
Marc Erickson, MathWorks, Natick MA, USA (*merickso@mathworks.com*)
Cristian Macario, MathWorks, Munich, Germany (*cmacario@mathworks.com*)
Giuseppe Ridinò, MathWorks, Turin, Italy (*gridino@mathworks.com*)

***Abstract*—As Moore's law is slowing down [1], enhancements of upcoming generations of integrated circuits (ICs) can't be primarily driven by the introduction of new technology nodes. Instead, most of the enhancements need to be extracted from the IC architecture, and the various components need to be optimized for the overall system architecture. This is making system-level design, simulation, and verification more and more important. Additionally, system-level verification allows engineers to find bugs early and shorten bug lead time, which ultimately leads to a better quality/time-to-market tradeoff. Traditionally, system-level verification tends to be overlooked, as it requires additional resources and effort. This paper outlines how this additional effort can be avoided by adopting a verification methodology that allows the reuse of system-level testbench components and verification scenarios (sequences) within UVM testbenches. As verification artefacts are reused throughout different design steps, the duplication of effort between system-level and RTL-level verification is reduced to zero.**

***Keywords*—System-level Verification; UVM; SystemVerilog; DPI-C; Reuse; Model-Based-Design; HLS**

I. INTRODUCTION

For a decade Moore's law has been slowing down. As a consequence, the enhancements (in terms of performance, cost per transistor, etc.) provided by the introduction of a new technology node are not sufficient to satisfy the IC market requirements. This means that additional improvements need to be achieved by optimizing the system architecture and the interaction between different components and domains (such as digital and analog). Therefore, engineers have been looking for workflows that facilitate system-level architectural exploration and that allow different domain specialists (such as system architects, designers, and verification engineers) to easily collaborate among each other. In fact, several related discussions are ongoing within the semiconductor community [2] [3] [4].

Enhancing system-level architectural exploration requires shifting part of the verification activities to the system level. This is often referred to as "Shift-Left" and it has the positive side effect of reducing bug lead time and improving the overall quality/time-to-market tradeoff. One drawback is that it can require additional resources and effort to be invested in verification at the earlier design stages. In this paper we describe an approach that allows engineers to set up system-level testbenches and associated stimuli that can be easily reused within a chip-level UVM testbench, eliminating duplication of effort. In the following sections, we will describe the application of this approach to the verification of an automotive voltage measurement IC.

II. SYSTEM-LEVEL DESIGN MODEL

The key component of the STMicroelectronics IC under study is called Main Measurement Unit (DUT in Figure I), which is modeled and verified using the Simulink platform. The model includes both analog and digital components. The rest of the IC components are either meant to ensure the correct operation of the Main Measurement Unit or they are managing the communication with the external world. As these functionalities are not key differentiators for the IC, they are not included in the system-level Simulink model.

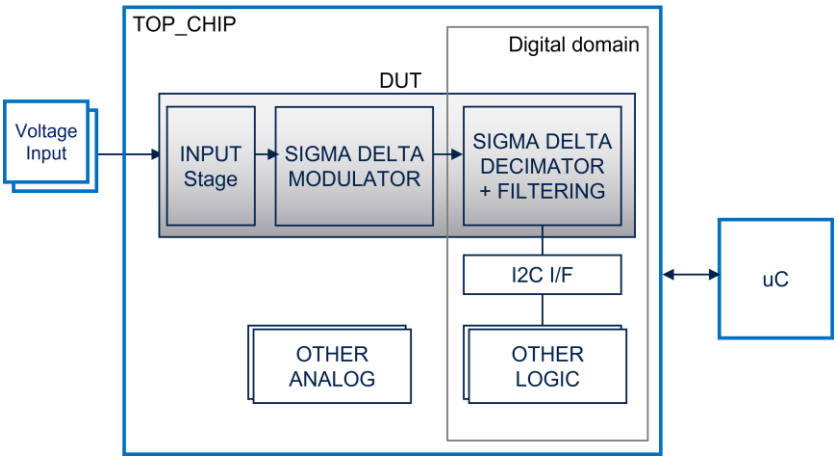

Figure I. Main Measurement Unit (DUT)

The voltage inputs of the IC are modeled in a behavioral way as ideal voltage generators. Such inputs are processed by the DUT, first by an analog input stage and then by a second order sigma-delta modulator. The analog part is modeled using real numbers, in discrete-time domain, with high sample rate (such as 1ns) compared to digital clock and analog time constants (in the order of us). The behavior of the analog Simulink models is aligned with the behavior of the corresponding SPICE netlist, at least in typical conditions.

The digital part of the DUT, including sigma-delta decimator, filtering, and a digital correction part, is modeled in a fixed-point, bit-accurate, and cycle-accurate way. This approach enables automatic RTL code generation directly from the model. It also saves coding time and establishes a common source of the design between system architecture and design implementation stages (see section V for detail). The digital sample time (corresponding to the clock period) is chosen considering the tradeoff between the power dissipation and the conversion time.

After the sigma-delta conversion, the data is filtered by a programmable average calculation, applied on successive conversions, also modeled within the Simulink platform as part of the DUT. The number of averages is a configuration input of the DUT (2^N samples). Finally, the average measurement is serially read by a microcontroller (uC) through an I2C interface, implemented with ST standard circuits not modeled within the Simulink environment. This application requires high accuracy and resolution (more than 10 bits) combined with a digital correction to compensate for the analog error.

III. SYSTEM-LEVEL VERIFICATION

The DUT is part of a higher-level Simulink model, which acts as a testbench (Figure II) to ensure that the design is functioning as expected as it is being developed, and to improve the quality of the design before it progresses to any implementation stage. This testbench includes the *Stimuli* block, which drives configuration and voltage stimuli, and the *Scoreboard*, which monitors the converted data and checks the results against the ideal conversion.

Figure II. Simulink Testbench

The Stimuli block (Figure III) drives the following DUT inputs:

- Vin signal, which models the voltage analog input with a real number.
- Configuration parameters, such as ADC enable, and configuration of the average filter. In the chip-level environment, these parameters are stored in the register map and set up using the I2C bus interface (not modeled in Simulink).
- Digital correction, which is used to compensate ADC errors. This information, in the chip-level environment, comes from an external NVM memory.
- Test mode configuration, which is used to set up the debug mode.

The Stimuli block generates simple test cases in which all the DUT inputs are parametrizable constants except for the Vin and the ADC's enable signal. All stimuli use a faster sample rate than the DUT; therefore they are connected by using *rate transition* blocks.

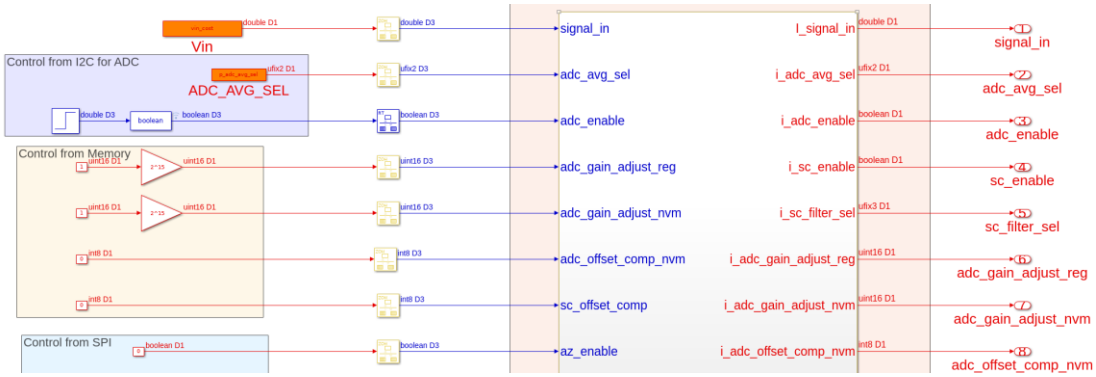

Figure III. Simulink Stimuli Block

The scoreboard (Figure IV) is the block used to perform the functional checks in the model simulation. It receives as input the ADC conversion and data valid signals, coming from DUT. The scoreboard consists of four main blocks:

- *Sampler*, which samples the DUT output data every time a new data valid is set.
- *Check Valid*, which controls the data valid signal and discards the first pulses. It generates a check_valid signal.

- *Check Range*, which compares the output of *Sampler* block, the DUT output data, with a range defined by the quantized Vin signal and the expected maximum uncertainty. It generates a logic value that is set when conversion is outside range.
- *Check Assertion*, which evaluates that data is correct when check_valid is true by using an Assertion Simulink block that reports the failures.

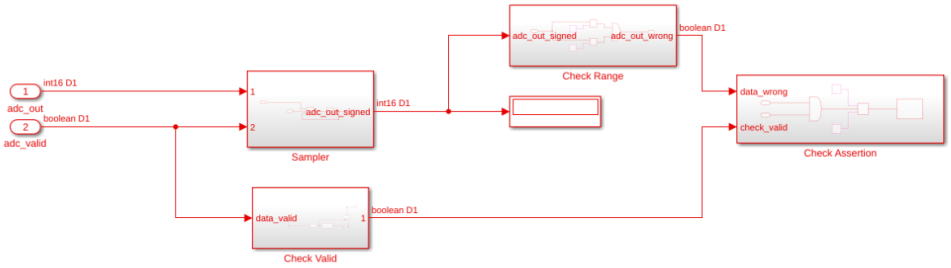

Figure IV. Simulink Scoreboard

IV. REUSE OF SYSTEM-LEVEL TESTBENCHES AND TEST CASES WITHIN UVM CHIP-LEVEL VERIFICATION ENVIRONMENT

The workflow described up to this point allows the engineer to shift-left the unit-level functional verification by leveraging the modeling capabilities of Simulink. In other words, the DUT (the Main Measurement Unit in the described case) can be verified before its corresponding RTL implementation (for the digital part) and SPICE netlist (for the analog part) are available.

As verification signoff needs to be done at implementation level, the system-level verification test cases designed and run in Simulink need to be re-run at RTL/SPICE level. In order to avoid duplication of effort, the MATLAB and Simulink platform offers several ways to reuse models and testbenches within both logic and SPICE simulators. Since the focus of the described activity was on the digital part of the design, and since the RTL verification environment was based on the UVM framework, it was decided to reuse the Simulink testbench using the Simulink UVM generation capability [5], that allows engineers to generate a UVM testbench from the Simulink environment. Such capability is based on the uvmbuild() MATLAB command provided by the HDL Verifier tool [7][8]. This command takes three arguments: DUT, stimuli, and scoreboard. Each of these three entities corresponds to a Simulink block (as shown in Figure II).

This MATLAB command generates a SystemVerilog UVM testbench equivalent to the Simulink one, where the DUT is implemented as behavioral model (generated as well). As previously mentioned in the case study, the Simulink DUT corresponds to the Main Measurement Unit, which is the main IC component (see Figure I), but it doesn't correspond to the full IC. Since the main RTL verification is run at chip level, it was then necessary to connect the UVM components generated from Simulink (mainly the Stimuli and the Scoreboard) within the UVM chip-level verification environment, so that they can be integrated within the chip-level regression. This includes test cases that are not generated from Simulink.

The digital chip-level testbench environment (Figure V) is composed of the chip-level top instance (TOP_CHIP) and a class-based chip-level UVM testbench. TOP_CHIP and testbench are connected by interfaces, which have the function of driving and monitoring the TOP_CHIP pinout. The main part of the verification environment is made of sequences and the UVM environment; each sequence representing a different test case, which generates transactions (sequence items) to be sent to the UVM agents. The UVM environment is formed by these UVM agents; whose role is to send the transactions (sequence items) generated by the sequences to the corresponding driver, which sends them on the interface, taking care of the timing and protocol aspects. Moreover, each UVM agent checks the data from its interface using a monitor component.

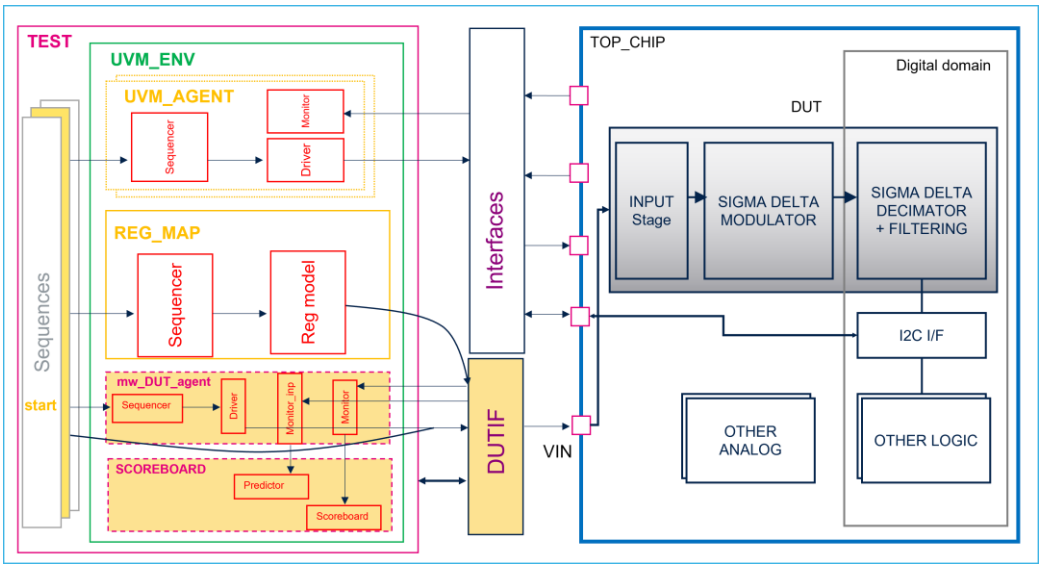

Figure V. Chip-Level UVM Testbench

As previously mentioned, the UVM components generated from Simulink by HDL Verifier [8] have been integrated in the described chip-level UVM testbench (components with yellow background in Figure V). These generated UVM components are:

- **Sequence**: The Stimuli block, which generates transactions
- **mw_DUT_agent**: Component that drives the transactions on the DUTIF interface. It also monitors the data coming from the chip-level top (TOP_CHIP) through the DUTIF interface. The component contains two monitor blocks: "classic" monitor and "input" monitor. The classic monitor simply translates the output data from the chip-level top into sequence items. The input monitor watches the DUT inputs and sends them to the scoreboard predictor, which uses it to compute the expected DUT output data.
- **Scoreboard**: Component that makes the comparison between the chip-level output (translated into sequence items by the monitor) and the expected output, computed using the inputs that have been sent to the DUT; for this scope, it is composed of a predictor and scoreboard.
- **DUTIF**: SystemVerilog interface generated from Simulink. In the case under study, only the Vin signal of this interface has been connected to the chip-level top. The rest of the signals correspond either to the configuration inputs of the Main Measurement Unit (see Figure III) or to the Main Measurement Unit outputs. Such signals do not exist as chip-level inputs, but only as internal signals that can be accessed by a I2C bus interface or by an NVM. Therefore, instead of being connected to the chip-level top, they are used to connect the UVM components generated from Simulink with the rest of the chip-level UVM testbench – through a top test sequence that works as interface decoder.

The top test sequence, as mentioned, allows the engineer to integrate the generated UVM components with the chip-level testbench, especially the drivers and monitors used to set and get the values of Main Measurement Unit inputs/outputs that are not exposed at chip level.

In order to perform this role, this test sequence observes the DUTIF and, whenever the value of a Main Measurement Unit configuration input gets updated, it generates and executes the (I2C or NVM) transaction that produces the same effect at chip level. Similarly, whenever a Main Measurement Unit output is supposed to have been updated, it generates and executes the bus transaction that allows the engineer to read its value and sends it to the scoreboard (through the corresponding DUTIF signal). In this way, it is feasible to integrate UVM generated unit-level components into a UVM chip-level testbench, without affecting the DUT.

Moreover, a manually inserted trigger has been introduced to ensure that the UVM sequence generated from Simulink is executed after the chip initialization has finished.

V. REUSE OF DUT MODEL: RTL GENERATION

The workflow described up to this point has been further improved by generating the RTL code of the digital components described within the Simulink environment, so that both Simulink testbench components and design models are reused at chip level.

Before introducing the RTL generation (enabled by the HDL Coder add-on to Simulink [9]), the digital components were modeled in Simulink at a higher level of abstraction, sometimes using real numbers, leaving the fixed-point or bit-level conversion to the RTL coding step to be done by hand. This way, such components were modeled twice: once at behavioral level and once at RTL level, with a corresponding duplication of effort, dependent on the block complexity. The verification of the hand-written RTL code was very time-consuming compared to the Simulink model verification, as most of the test cases had to be specifically designed for the RTL level. Introducing RTL code generation allowed block-level RTL verification to be avoided, focusing on chip-level instead, as the block-level verification is performed within Simulink. Therefore, other than improving the design efficiency and quality, the introduction of HDL Coder also enabled a better balance of the verification activities between Simulink and traditional digital simulations.

```
-- -------------------------------------------------------------
--
-- File Name: C:\hdlsrc\vm_model\adcv_corr.vhd
--
-- Generated by MATLAB 9.9 and HDL Coder 3.17
--
-- -------------------------------------------------------------


-- -------------------------------------------------------------
--
-- Module: adcv_corr
-- Hierarchy Level: 1
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY adcv_corr IS
  PORT( i_clk                             :   IN    std_logic;
        i_reset_n                         :   IN    std_logic;
        i_adjust                          :   IN    std_logic_vector(15 DOWNTO 0);  -- uint16
        i_adc                             :   IN    std_logic_vector(19 DOWNTO 0);  -- sfix20
        i_comp_dis                        :   IN    std_logic;
        o_adc_cal                         :   OUT   std_logic_vector(19 DOWNTO 0)  -- sfix20
        );
END adcv_corr;


ARCHITECTURE rtl OF adcv_corr IS

  -- Signals
  SIGNAL i_adc_signed                     : signed(19 DOWNTO 0);  -- sfix20
  SIGNAL i_adjust_unsigned                : unsigned(15 DOWNTO 0);  -- uint16
  SIGNAL Product1_cast                    : signed(16 DOWNTO 0);  -- sfix17
  SIGNAL Product1_mul_temp                : signed(36 DOWNTO 0);  -- sfix37
  SIGNAL Product1_out1                    : signed(35 DOWNTO 0);  -- sfix36
  SIGNAL Extract_Bits1_out1               : signed(34 DOWNTO 0);  -- sfix35
  SIGNAL Gain1_cast                       : signed(35 DOWNTO 0);  -- sfix36_En15
  SIGNAL Gain1_out1                       : signed(19 DOWNTO 0);  -- sfix20
  SIGNAL Switch8_out1                     : signed(19 DOWNTO 0);  -- sfix20
  SIGNAL Unit_Delay6_out1                 : signed(19 DOWNTO 0);  -- sfix20

BEGIN
  -- Rescale down to the original fixed point
  -- and truncate

  i_adc_signed <= signed(i_adc);

  i_adjust_unsigned <= unsigned(i_adjust);

  Product1_cast <= signed(resize(i_adjust_unsigned, 17));
  Product1_mul_temp <= i_adc_signed * Product1_cast;
  Product1_out1 <= Product1_mul_temp(35 DOWNTO 0);

  Extract_Bits1_out1 <= signed(Product1_out1(34 DOWNTO 0));

  Gain1_cast <= resize(Extract_Bits1_out1, 36);
  Gain1_out1 <= Gain1_cast(34 DOWNTO 15);


  Switch8_out1 <= Gain1_out1 WHEN i_comp_dis = '0' ELSE
      i_adc_signed;

  Unit_Delay6_process : PROCESS (i_clk, i_reset_n)
  BEGIN
    IF i_reset_n = '0' THEN
      Unit_Delay6_out1 <= to_signed(16#00000#, 20);
```

Figure VI. Snapshot of the Generated RTL Code

Design quality has shown improvements, since the generated RTL is bit-true, cycle-accurate, and synthesizable. Human errors in converting Simulink models to RTL are avoided, ensuring perfect alignment between Simulink and RTL.

Using RTL generation, most of the bugs can be discovered during architectural exploration and optimization, bringing forward the verification in the flow. Once a bug was found at RTL level, in the previous flow, it had to be fixed separately at RTL level and at Simulink level. This required new simulations at both levels to validate it, doubling the number of design iterations needed. By contrast, when generating the RTL automatically, bug fixes are only needed at the Simulink level, and they can be verified at the same level. After verification, the HDL code can be re-generated, ensuring alignment between Simulink design models and RTL. As one can imagine, this implies a significant improvement in the workflow efficiency.

VI. RESULTS AND FUTURE ENHANCEMENTS

The described design and verification approach allows engineers to save time by avoiding duplication of effort between system-level and RTL-level verification. It also allows engineers to have a single source of test cases for the stimuli that are run at both system- and RTL-level. As a consequence, the RTL verification time has been cut in half. In addition, the ability to generate RTL code from the Simulink design models also allows engineers to save coding (design) time, reduce the chance of human errors, and use the same design source for system and RTL simulation, thus further reducing the IC design effort.

In order to more easily integrate the generated UVM components within a chip-level testbench environment, MathWorks and ST are exploring the use of the Register Abstraction Layer and more modularization options of interfaces to UVM agent hierarchies. These will allow abstract models from MathWorks tools to be more easily integrated within the chip-level design verification framework.

In the described case study, for simplicity, the various test cases have been run sequentially, one after the other, as part of the same verification sequence. This approach is very straightforward but lacks scalability. To improve this and set up an environment where each test case can be run and debugged independently from the others, two approaches can be taken:

1. Set up several Simulink testbenches, one per test case, run them in Simulink, and then generate UVM sequences from them.
2. Use Simulink parametrization features to make the Simulink testbench capable of generating all the needed test cases, depending on the values of the parameters.

To implement approach 1, one can leverage the features offered by various Simulink add-ons: test harnesses (offered in Simulink Test), that allow users to set up different testbenches verifying the same DUT, and the test manager, that allows users to run all test harnesses (test cases) and manage the regression. Moreover, it is possible to collect and analyze coverage results within the Simulink environment by leveraging the capabilities of the Simulink Coverage add-on. Approach number 2 leverages Simulink parametrization and code generation capabilities to implement versatile test cases. One can define a set of parameters that affect the stimuli generated by the testbench. Such parameters can be tuned both within the Simulink environment and within the UVM environment.

Approach 1 is suitable when the test cases to be applied to the DUT are very different from each other, whereas approach 2 is convenient when test cases are similar to each other, as each test case shares the same stimuli generation infrastructure. In many cases the most efficient strategy would be to apply a mix of approach 1 and 2, in order to leverage the advantages of both of them.

Often it is useful to introduce randomization within testbenches and test cases. There are mainly two ways to introduce randomization to the described workflow:

1. Use Simulink randomization blocks and/or MATLAB randomization functions.
2. Introduce the randomization at UVM level, after the UVM testbench has been generated from Simulink [6]. Randomization can act on transactions (sequence items) as well as on testbench parameters.

The first approach allows one to randomize the stimuli at system level, within the Simulink platform, but it offers limited capabilities in terms of constraint management and solving. The second approach, instead, allows one to exploit the full SystemVerilog and UVM randomization capabilities. But, of course, this approach can't be used to randomize the test cases at Simulink level, that, in this case, are used as "parent" test case, with randomization and corresponding constraints added to the generated UVM sequence.

VII. CONCLUSION

The described approach allows engineers to shift-left the untimed and loosely timed portions of IC and IP verification to system-level and to reuse the system-level design models and verification environments within the chip-level RTL environment, thus avoiding duplication of effort. This approach is particularly valuable in the verification of ICs that use complex algorithms and/or ICs where different domains, such as analog and digital, are tightly interacting with each other. The approach allows engineers to easily model different domains and to find, early in the design flow, functional and performance issues in algorithms, state machines, and the interactions between different domains (such as digital, Analog, RF, and SW).

REFERENCES

[1] Tom Simonite, "Moore's Law Is Dead. Now What?", MIT Technology Review, May 13, 2016

[2] Mark Burton et al., "Hybrid System Simulation Standards", DVCon Europe 2020, October 27, 2020

[3] Manfred Thanner, Ingo Feldner, Sacha Loitz, Ralph Schleifer, Kevin Brand, "Automotive Virtual Prototypes", DVCon Europe 2020, October 27, 2020

[4] Tom Fitzpatrick, Prabhat Gupta, Matan Vax, Karthick Gururaj, Hillel Miller "Portable Stimulus: What's Coming in 1.1 and What it Means For You", DVCON U.S. 2020, March 2, 2020

[5] MathWorks documentation, "UVM Generation", https://www.mathworks.com/help/hdlverifier/uvm-generation.html, 2021

[6] MathWorks documentation, "Add Random Constraints to Sequences in UVM Test Bench", https://www.mathworks.com/help/hdlverifier/ug/uvm_add_random_constraints_to_sequence.html, 2021

[7] MathWorks documentation, "uvmbuild", https://www.mathworks.com/help/hdlverifier/ref/uvmbuild.html, 2021

[8] MathWorks documentation, "HDL Verifier", https://www.mathworks.com/help/hdlverifier/, 2021

[9] MathWorks documentation, "HDL Coder", https://www.mathworks.com/help/hdlcoder/, 2021