2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Methodology for Automated Generation and Validation of Analog Behavioral Models for Mixed-Signal Verification

Nan Ni, Infineon Technologies AG, Munich, Germany

Chunya Xu, Infineon Technologies AG, Munich, Germany

Sebastian Simon, Infineon Technologies AG, Munich, Germany

Email:<Firstname.Lastname>@infineon.com

***Abstract*—As the complexity of mixed-signal SoC designs keeps growing nowadays, top-level simulation is inevitable to guarantee the correct functionality. While the high accuracy of SPICE-models is not necessary for top-level verification, they are replaced with analog behavioral models to accelerate the top-level simulation. However, the creation and validation of analog behavioral models still imply much manual work. As a real number model shows advantages in simulation speed and digital integration, we propose an approach in this paper to generate real number models of analog components based on the Infineon's meta-modeling concept. Meanwhile, the automation of UVM-MS testbench generation for model validation is also presented in this paper.**

Keywords—UVM; analog behavioral model; mixed-signal; real number model

I. INTRODUCTION

In the mixed-signal SoC designs, digital and analog modules become so intertwined that the previous verification method, importing pre-verified digital/analog IPs into an analog/digital design, is no longer adequate. To run the top-level simulation, simulation speed is a significant factor which influences the whole product cycle. Real number modeling is a technique to model analog blocks that allows the simulation to run in a pure digital environment, which to a great extent shortens simulation time compared to the accurate SPICE model.

We introduce a method to automate the model generation and validation flow. Based on the Infineon's meta-modeling framework, the linear or nonlinear Real Number Model (RNM) and its accompanying UVM-MS testbench can be generated from the user specifications.

II. RELATED WORK

To create a real number model, the manual writing and scripting methods were mostly used previously, such as writing a VHDL real number model of filters with operational amplifier according to the transfer function [1], or using scripts to identify blocks from existing netlists and replace them with real number models [2]. Moreover, before a model comes into use, it has to be validated to make sure that it delivers the same correct functionality as the circuit implementation. Methods which are used currently such as in-system co-simulation and side-by-side simulation [2] still require manual inspection.

A transaction-oriented UVM-based library, known as A-UVM, contains analog features to generate constrained-random analog stimulus and trigger analog transactions, uses metrics to compare the analog transactions, and supports functional coverage [3][4]. Based on it, an analog UVM testbench can be created to compare a real number model with the SPICE realization. However, constructing an A-UVM testbench still requires manual work in writing test cases and checkers.

In order to automate the model generation and validation flow, different scripting techniques, as well as code generation techniques are developed. Infineon's proprietary metamodeling framework targets generating codes from abstract specifications [5]. Given a defined meta-model in the UML diagram, an infrastructure can be generated with APIs to communicate with the model structure and the generated view. A GUI is also provided to enter the user-defined parameters for the instance generation. Target code transformations are coded in Python and Mako templates, which provides the user with possibilities of generating different SW/HW languages.

Previous approaches of RNM focus either on the generation or the validation of models. However, the novelty of our approach is the adoption of meta-modeling concept to realize the automation of the whole generation and validation flow based on the single source. From defining the specification of RNM and TB to getting the validation report, only little manual work is required.

III. Generation And Validation Flow

A. *Overview of Generation Flow*

In this work, a methodology to realize the automated generation and validation of the real number model is proposed. As shown in Fig. 1, we cover both linear and nonlinear components in the generation flow. The generation of the validation testbench is based on the real number model. It aligns with the specification of the RNM, meaning that the generation of the testbench adapts to any changes in the setting of a component such as adding a port or changing a parameter. The flow also includes the generation of a run script which is able to adapt to different simulators and include the path of used packages automatically.

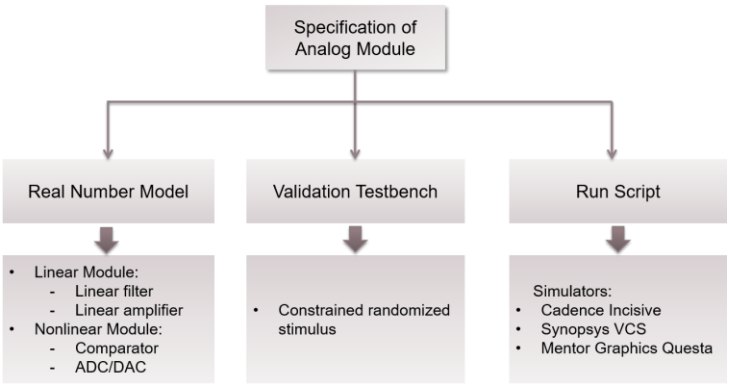

Fig. 1. Framework of the Generation Glow of the Real Number Model, Testbench and Run Script

The whole generation flow utilizes Infineon's proprietary metamodeling framework to generate design and verification codes based on defined meta-models. We create the meta-model of the real number models of analog components, define the meta-model of an A-UVM testbench, and write the accompanying Mako templates for code generation. The metamodeling framework contains a GUI for parameter input. Using different user specification input for the meta-model, a project-specified model can be flexibly created with the framework.

Fig. 2 demonstrates a simplified top-level view of the UML diagram, which is the meta-model. The left part of the root node refers to analog modules including both linear and non-linear modules. The right part of the UML diagram corresponds to the A-UVM testbench. It includes both digital and analog features, such as digital/analog sequence, agent, and scoreboards. The testbench generation refers to the parameter setting of the analog module. For example, the number of analog ports determines the number of analog sequences to be generated.

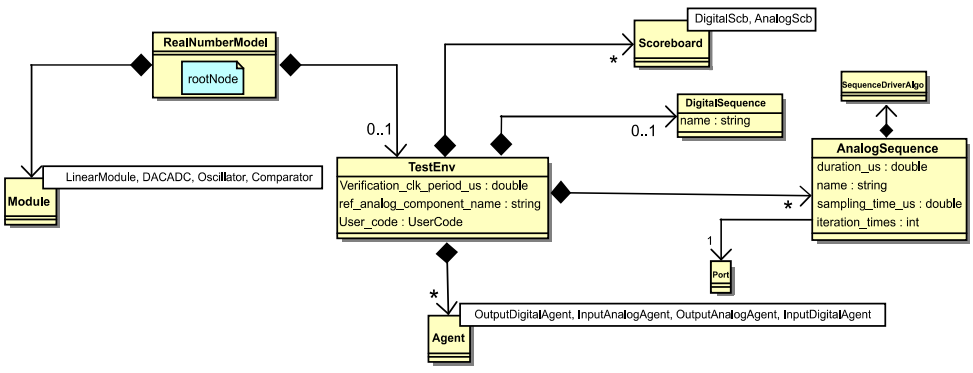

Fig. 2. (Simplified) Top-Level View of UML Diagram for Analog Module and Testbench Generation

B. *Linear and Nonlinear Model Generation*

Metamodeling framework's concept is to generate views from the user specification. In our work, for a linear system, only the poles and zeros are the necessary inputs, from which the SV behavioral models can be generated. For a nonlinear component, the key parameters which determine the functionality of the component work as the input of the metamodeling framework, such as the reference voltage and resolution of an ADC or DAC.

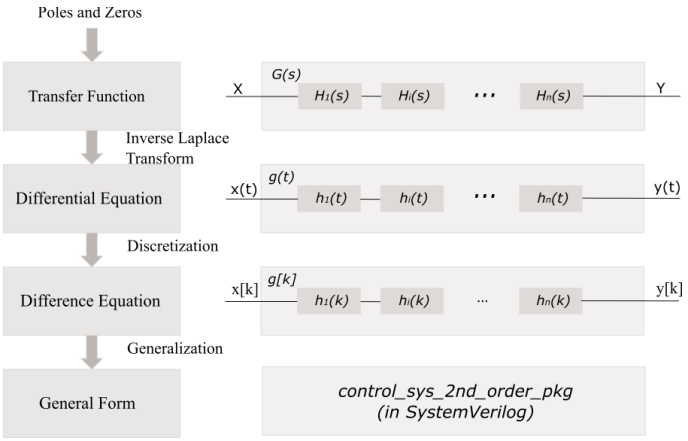

Fig. 3. Mathematical Background of Generating a Linear Module

Fig. 3 explains how the system can be interpreted step by step from defining poles and zeros to the final SystemVerilog RNM representation. A transfer function in S domain can always be converted into a pole-zero representation as shown in Equation (1), which acts as the parameters that the metamodeling framework takes for the generation.

$$G(s) = k\frac{\prod_{m=1}^{M}(s-z_m)}{\prod_{n=1}^{N}(s-p_n)} = k\prod_i H_i(s), \quad M \le N \tag{1}$$

Inverse Laplace Transform is applied to the transfer function to calculate the differential equation. With the differential equation, the next step is to get the difference equation by discretization using approximation method like forward Euler method, backward Euler method or Trapezoidal rule. Our work basically uses the forward Euler method to calculate the difference equation. With the difference equation, we can get the response signals iteratively according to the defined time step. As the transfer function of a complex linear system may have more than one pole and zero, we can decompose the function into subsystems like the Equation (2).

$$\mathrm{G(s)} = \frac{Y(s)}{X(s)} = H_1(s)\cdot H_2(s)\cdots H_i(s)\cdots H_n(s) \tag{2}$$

All of the subsystems can be divided into 6 groups, each of which has different combinations of the number of poles and zeros. In each group, the pole or the zero has either real or complex values, forming different types of a subsystem such as a subsystem with only one real-number zero and two poles of complex numbers or a subsystem with no zero and one real-number pole. As a result, the subsystems can be regarded as the basic blocks of the complicated transfer function. In another word, all the transfer functions can be represented by the combination of subsystems in these 6 groups. The discrete formats of these subsystems are calculated in advance and included inside a pre-written SystemVerilog package named *control_sys_2nd_order_pkg*. In conclusion, any transfer function of the linear system can be decomposed into subsystems and expressed using the discrete format in the package iteratively. The only step that the user has to prepare to generate a linear module is the values of poles and zeros. The rest steps can be achieved with the metamodeling framework and the SV templates.

Fig. 4 involves the part of the UML diagram of the linear and nonlinear modules. The transfer function parameters are the most important building blocks composing the linear module. Each composed UML class of the transfer function parameter corresponds to an input in the GUI, meaning that the user can define poles and zeros in

either real or complex number in the GUI, which are taken by metamodeling APIs as arguments and be generated in the SV codes. The meta-model also includes features such as output saturation and clock deactivation.

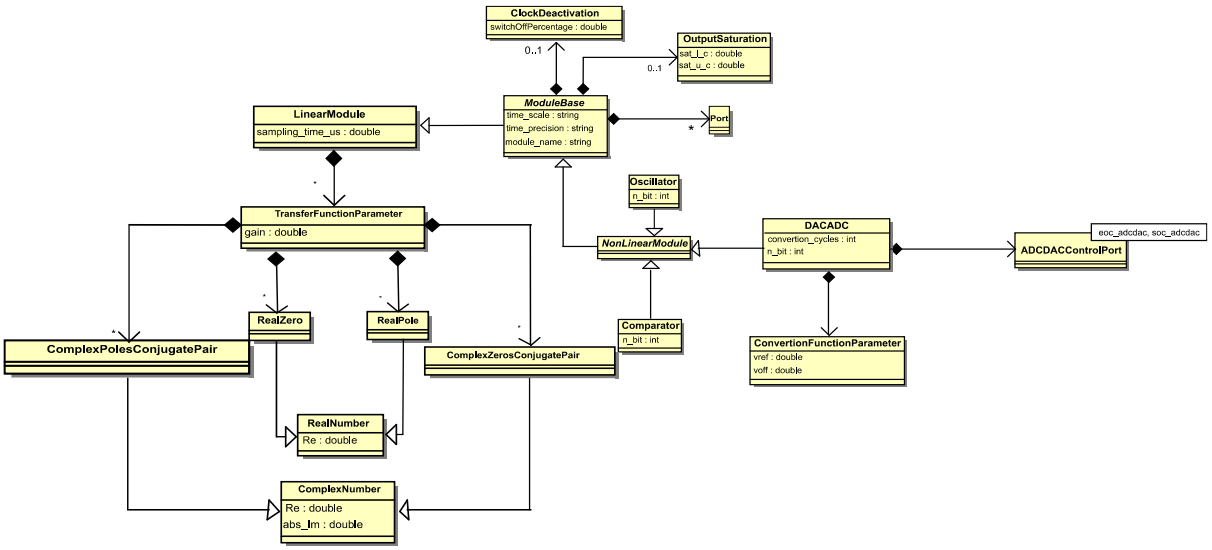

Fig. 4. (Simplified) Part of UML Diagram of Module Generation in the Generation Flow

```
initial beign
        my_sys = new;
        cons = my_sys.construct(z_complex_vec,z_real_vec, p_complex_vec, p_real_vec );
end

always @(posedge clk)  begin
        vin_s = vin;
        resp = my_sys.get_response(vin_s, T);   //response calculation
        vout_s = resp * gain;
end
```

Listing 1. Part of the Generated Code of a Linear Module

Listing 1 shows the example code of a generated RNM of the linear filter. It takes the vector of poles of zeros as parameters from metamodeling framework GUI. The get_response function calculates the response which applies the aforementioned calculation method and returns with the result.

The generation of the nonlinear module is similar to the generation of linear modules. In Fig. 3, among all the typical nonlinear components, the ADC and DAC are implemented in our flow. However, due to the time limit, other non-linear components in the plan such as oscillator and comparator are not implemented so far.

Similar to the transfer function for linear systems, the conversion function is the key to realize an ADC/DAC. To generate an ADC/DAC, the deciding parameters are the reference voltage and the resolution of the conversion, which are the input of the GUI that the user can configure. With the input of necessary parameters, the target code can be generated with the Mako template.

```
Zmax = $pow(2, n_bit);
xin_0 = adc0_i_p -  adc0_i_p;
Dout_zw_0 = Zmax * (xin_0 + vref)/(vref + vref);
```

Listing 2. Example of the Generated Code of an ADC

A generated example code of an ADC is shown in Listing 2 describing the function between the differential analog input and the digital output according to the reference voltage and converting resolution.

For both linear and nonlinear modules, there is a port structure defined in our meta-model, which means that the user can select the type of port: select whether it is analog or digital, define the bit of digital port, or select whether it is differential of an analog port, and define the name of port in the metamodeling GUI. The ports can be automatically generated in the entity in the behavioral model, and are also connected with the testbench generation process.

C. UVM-MS Testbench Generation

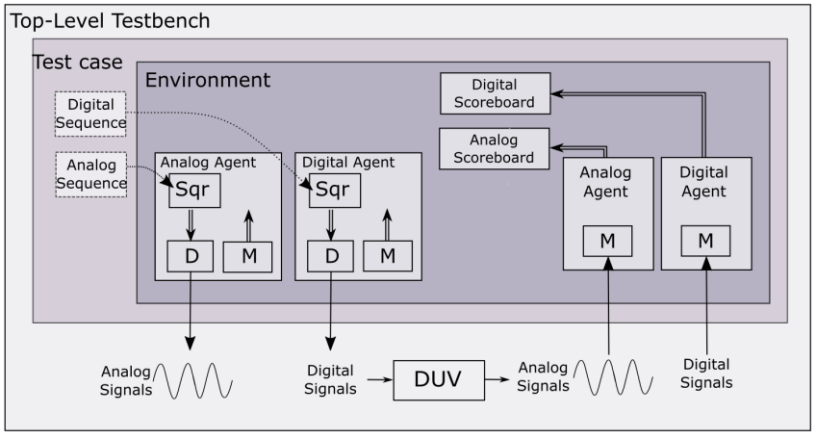

Fig. 3. A Generic UVM Testbench with Both Analog and Digital Features

Fig. 3 shows the structure of a generic form of the generated testbench supporting both analog and digital features. All the features are contained in the top-level meta-model UML diagram in Fig. 2.

The following necessary UVM components make up a UVM test environment: the agent, the scoreboard, and the sequences started in the test case.

1) Testcase and Sequence

To validate the generated component, a test should include the digital sequence or analog sequences according to the type of input data of the component. For example, an ADC has an analog input hence it needs an analog sequence while a DAC has a digital input which needs a digital sequence. Once a digital sequence is selected to be generated in the metamodeling GUI, it automatically bounds all digital inputs together, generates the corresponding digital sequence item class and the sequence randomizing and sending out the transactions. On the other side, as for analog signals, it cannot be directly randomized using the existing UVM features. So A-UVM makes it possible to generate randomized real-type data using the functions from the A-UVM library.

There are five types of common data structures supported by A-UVM library: ramp, sine, constant, sampling, and Fourier. For each data structure, there are also corresponding driver and monitor algorithms to send and collect the same type of data transactions. Each data structure has its typical parameter(s), such as level and increment for ramp data structure. These parameters are set by the user in GUI and can be randomized in the process to generate different shapes of waveforms. Each analog port has one specific analog sequence, which means that different ports can have different data structures. To set up one analog sequence, the user only needs to choose the port and data structure, define the name, duration, sampling time and iterations.

2) Environment and Agent

A UVM environment is the place where active and passive UVM agents are instantiated. As the A-UVM library provides the user with a general agent class which offers interface functions for configuration on the hierarchical upper class, the user can set everything inside the environment after instantiating the agents. An active agent contains an A-UVM sequencer, a driver, and a monitor while a passive agent only contains an

A-UVMM monitor. In the GUI, the user can choose from analog/digital active/passive agents to instantiate according to different components.

In a pure digital verification environment, we only need one digital active agent to bound all input signals and one passive agent for all output signals for a simple test case because one transaction is able to contain all necessary digital data information. On the contrary, as each analog port may involve a different type of waveforms or different data structure, each port needs a separate analog agent to employ individual driving and monitoring algorithms. Once the digital agents are selected, the user does not need to define anything for the digital sequencer, driver and monitor classes, hence they can be generated automatically with the Mako file for driving and monitoring all selected digital signals.

The analog agent shows more complexity as it uses features enabling analog behaviors. As a result, the user needs to configure more parameters for it. Unlike the digital transactions, analog transactions need a special triggering mechanism to trigger transactions according to a certain feature. So far, seven types of built-in triggers are realized in the generation flow in this work such as time trigger, level trigger or event trigger. The user can select the type of trigger which is necessary to be set up for the monitor inside the analog agent. For instance, if we set up a testbench for comparing the output of a linear filter and its reference model, we can trigger one transaction every 10 ns uniformly. This can apparently be achieved by a time trigger. After setting the time trigger to be 10ns, transactions are triggered every 10ns and be sent to the scoreboard for further comparison.

3) Scoreboard

Considering different types of modeling components such as ADC, DAC and the linear filter, the signals from output ports can be either digital or analog. For ADC and other components which have digital outputs, the comparing target is the digital transactions from digital monitors for the behavioral model and the analog component. This can be achieved by bit-wise comparison of different signals between transactions. Nevertheless, it is no longer practical to do a bit-wise comparison for analog output signals. Therefore, specific algorithms are applied for the comparison between two analog transactions.

The idea is to instantiate one or more scoreboards for corresponding analog or digital output transactions, wherein each analog output signal needs one dedicated scoreboard and all digital signals share one digital scoreboard. Differently, in the UML diagram, an analog scoreboard has a reference to output analog agent meaning that each analog output has a scoreboard with a comparison algorithm chosen from Cosine Similarity, Pearson product-moment correlation coefficient, Tanimoto Distance and Earth Mover's Distance. Furthermore, since the result of the comparison is always a value between “0” to “1”, a tolerance should be set to indicate a fail or pass of the test case. Therefore, to generate the scoreboard, the user only needs to select the comparison algorithm and sets the tolerance for the selected analog output port in the GUI. After running the simulation, the comparison between the result and the tolerance shows whether the validation result.

D. Run Script Generation

Current generation flow only implements Cadence Incisive simulator in the automation process. As the generation of the run script runs together with the generation of behavioral model and the testbench, it iterates the packages such as complex number packages, as well as DPIs such as C functions used in the testbench, thus the path can be included into the script automatically. The generation flow can be extended to support other simulators.

This generation flow makes the creation, validation and simulation execution in an automatic way. Once the specification of an analog component is ready, the real number model can be generated using this flow with/without the validation testbench. This means, on the one hand, the generated model can be used on the top-level verification before the analog design is ready, to avoid delaying the top-level test cases focusing on other sub-blocks to gain an early-stage result. On the other hand, the model can be used after analog IP is ready, meaning that it simulates with

the validation testbench, gets validated, and finally replaces the analog IP in the top-level mixed-signal design to shorten verification time.

IV. Application On A Real Project

We apply our approach on a project of a radar chip, where the modeling target is the Multi-ADC (MADC) in the radar chip. In the GUI, port definition, the number of ADC channels, ADC conversion resolution, and the reference voltage are defined by the user exactly the same as the MADC in the radar project. To generate the testbench, as the MADC has two channels, meaning that there are two analog inputs, two analog sequences can be defined in GUI. In this project, we select ramp signals as input. Since the output is digital signals, there is no need to set the analog comparison algorithm in the scoreboard.

The generated real number model of the ADC shows a speed-up of around 16 times compared to the previous VHDL behavioral model in the project. The simulation time of RNM is 4.5s while the VHDL implementation of MADC consumes 74.1s. Besides, we also encountered a bug in the actual analog implementation represented by a deadlock in the digital control logic, as shown in Fig. 4 and Fig. 5. The digital output in the implementation of MADC is stuck at "FFE" after the deadlock is encountered. Our generated testbench is capable of finding this bug together with the generated model. In Fig. 5, the bug in the design is fixed, so there is no longer UVM-Error claimed. This is more efficient and systematic than manual supervising of waveforms. Meanwhile, we follow the four-eye principle which makes sure that the generated model does not inherit potential bugs in the analog implementation.

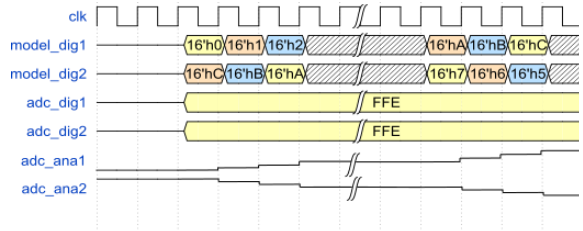

Fig. 4 Error Detected Waveform of MADC

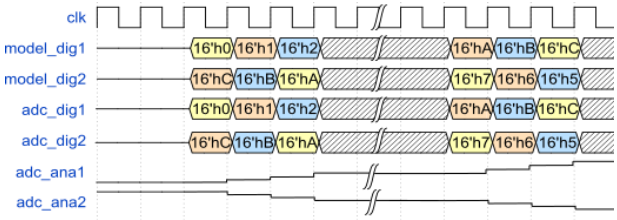

Fig. 5 Error Fixed Waveform of MADC

V. Conclusion

This paper introduces a flow to automate the generation and validation of a real number model based on Infineon's meta-modeling framework. This automation flow shortens the model development time as well as validation time. Finally, the developed model can be applied to the top-level simulation which dramatically shortens the run time of mixed-signal simulation. However, this model generation flow only focuses on the basic functionality of linear and non-linear modules, though, there are still many practical features of analog circuits that should be taken into consideration. For future work, more practical analog features can be included in the RNM generation such as X value propagation and integration check via assertions. The current implemented model generation flow can be extended with more detailed specifications and more non-linear components. Furthermore, for UVM testbench, not all of the A-UVM features are covered in the generation flow, such as the analog coverage, which is included in the A-UVM library but not yet implemented in the flow.

References

[1] Real Number Models for OP AMP Filters implemented in VHDL, 2017 International Conference on Optimization of Electrical and Electronic Equipment (OPTIM) 2017 Intl Aegean Conference on Electrical Machines and Power Electronics (ACEMP).

[2] MacGrath, John, Lynch, Patrick & Boumaalif, Ali (2015): Comprehensive AMS Verification using Octave, Real Number Modelling and UVM, DVCON Europe 2015.

[3] Rath, A. W., Esen, V. & Ecker, W. (2014): A Transaction-oriented UVM-based Library For Verification of Analog Behavior, 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC), S. 806-811.

[4] Simon, S., Bhat, D., Rath, A., Kirscher, J. & Maurer, L. (2017): Coverage-Driven Mixed-signal Verification of Smart Power ICs in a UVM Environment, 2017 22nd IEEE European Test Symposium (ETS), S. 1-6.

[5] Ecker, W. & Velten, M. (2015): Automating Design and Verification of Embedded Systems using Metamodeling and Code Generation Techniques, Design and Verification Conference and Exhibition 2015, United States.