Leveraging Model Based Verification for Automotive SoC Development

Aswini Kumar Tata
Allegro Microsystems, Manchester, NH, USA
atata@allegromicro.com

Bhanu Singh
MathWorks, Natick, MA, USA
bsingh@mathworks.com

Sanjay Chatterjee
Allegro Microsystems, Manchester, NH, USA
schatterjee@allegromicro.com

Eric Cigan
MathWorks, Natick, MA, USA
ecigan@mathworks.com

Kamel Belhous
Allegro Microsystems, Manchester, NH, USA
kbelhous@allegromicro.com

Surekha Kollepara
Cyient, Hyderabad, India
Skollepara.cw@allegromicro.com

***Abstract*- Shift-Left testing is a term used in the software and silicon development cycles to start testing as early as possible. For hardware verification, Model Based Design (MBD) enables shift-left testing at a higher level of abstraction with significant competitive advantage to achieve productivity, efficiency, and cost reduction. We propose an effective and practical shift-left solution at the system level which fares against customer requirements and can then be reused at the chip-level Universal Verification Methodology (UVM) environment, resulting in finding and fixing bugs earlier.**

In this paper, we describe the Model Based Verification (MBV) methodology with an example of Model Based testbench developed for a Coordinate Rotation Digital Computer (CORDIC) DSP IP design. The intended audience of this paper is system engineers, algorithm engineers, design and verification engineers. We have organized this paper into five parts - Introduction, Related Work, Proposed Workflow, Example Bench and Conclusion.

Keywords: Model Based Verification, Simulink, MATLAB, UVM code generation, uvmbuild

I. Introduction

Allegro Microsystems develops advanced semiconductor technology for the implementation of application-specific algorithms that sense, regulate, and drive a variety of mechanical systems. Allegro's semiconductors provide functions such as sensing angular or linear position, driving electric motors or actuators, and regulating the power applied to sensing and driving circuits for safe and efficient operation. Automotive Manufacturers are adding new sensing and processing capabilities in vehicles. These are sophisticated mixed signal sensors. The requirements from customers are becoming more complex and time to market tighter, which is resulting in a more complex algorithm with a reduced time to market. The stringent quality requirements in the automotive domain present a significant challenge in terms of developing and verifying these designs. Also, these mixed signal sensor chips are being developed for multiple customers with some requirement variations. The digital part of the algorithm will need to interface with analog blocks as well as external mechanical systems (e.g., Coils) and, in some cases, there are feedback loops between digital and analog that adds more complexity. This is resulting in the need to move architecture and algorithm development up to a higher abstraction layer, thus helping in better definition and implementation of algorithms.

Many of these sensor chips interact with mechanical systems that themselves need to be modeled and mapped to an algorithm model of the chip for better understanding of the overall system. For Verification engineers, understanding the application side of the chip and injecting realistic stimulus is becoming much more critical in finding genuine design bugs. Critical verification time and simulator licenses are being wasted debugging unrealistic scenarios while verifying purely digital blocks in a mixed signal sensor. The later we find bugs in the project cycle, the longer the time and higher the cost and effort needed to implement and verify the fix. All these challenges are pushing the Design Verification (DV) community to shift their verification efforts to the model development level.

In this paper, we describe how to build model-based testbenches and perform early testing when design behavior is modeled in MATLAB, Simulink. Our approach highlights reusing the Simulink test environment for UVM bench development and extending the generated UVM bench to add more complex constrained randomizations, assertion checkers and cover groups. Collecting model-based coverage, dead logic detection, rollover and saturation checks helped us in finding corner case bugs in the design. In this way, DV engineers can leverage the benefits of exhaustiveness, that UVM typically brings in, and reusing the verification effort put in earlier while developing the Model based bench. Moreover, merging coverage – code and functional coverage – from the UVM test runs between algorithm blocks and non-algorithm blocks in RTL will be much simpler to deliver during coverage sign off, reducing the effort for the coverage closure phase. We also share our experience of building a stimulus model that can generate multiple test scenarios and how to replicate stimulus between model-based test runs and RTL test runs.

II. Related Work

A range of other techniques have been developed to support the development of high-level testbenches to enable earlier development of ASIC verification workflows. In 2013, MathWorks introduced automatic generation of SystemVerilog DPI-C from Simulink models and added support for SystemVerilog DPI-C generation in 2014 [1]. The SystemVerilog DPI component generator produces shared libraries from MATLAB functions or Simulink subsystems for specified arguments and data types. It generates a directory structure that includes DPI-C wrappers, header files, makefiles, SystemVerilog testbenches, and simulation scripts for several commercial HDL simulators.

Many subsequent approaches involved automatic generation of UVM environments. In 2012 Mentor Graphics introduced UVM Framework (UVMF) to aid design teams adopting the UVM methodology [2]. UVMF is an open-source package that provides a UVM methodology and code generator for rapid testbench generation [3]. Mentor Graphics subsequently updated UVMF to automatically incorporate DPI-C components generated from MATLAB and Simulink by this method [4]. Several years later, Mentor Graphics added UVM generation to the Catapult™ HLS product through integration with UVMF, with the HLS C models integrated as predictors [5].

In 2019 MathWorks added the ability to generate complete UVM environments from Simulink subsystems for various design topologies with subsystems corresponding to UVM sequences, drivers, DUTs, monitors, scoreboards, etc. [6].

In a 2021 DVCon Europe paper, STMicroelectronics engineers using Simulink developed a design and verification approach for mixed-signal ASICs that employed automatic UVM generation and RTL generation for the DUT [7]. They saved time by avoiding duplication between system-level and RTL-level verification, resulting in design efficiency and quality improvements. In a 2022 DVCon US paper [8], Silicon Labs engineers using MATLAB to design filter banks improved their ASIC design verification workflow by automatically generating UVM testbench components from MATLAB code.

This paper is based on the use of UVM generation from design specifications developed in MATLAB and Simulink as high-level modeling languages. Generation of RTL from Simulink is also discussed in the context of improving the overall design and verification workflow.

III. Proposed Workflow

In a traditional workflow (Fig.1), system architects hand off a specification document to the design-verification team, which is then used as a golden reference. In comparison, the MBD verification approach (Fig.2) is a significant improvement from a document-based specification approach. A Model acts as an executable specification to improve specification hand-off and captures design behavior at a higher level of abstraction. The capability to simulate the expected design behavior in the form of a model allows better understanding of expected behavior listed in the specification. The system architects, designers and verification engineers can then better collaborate using the models.

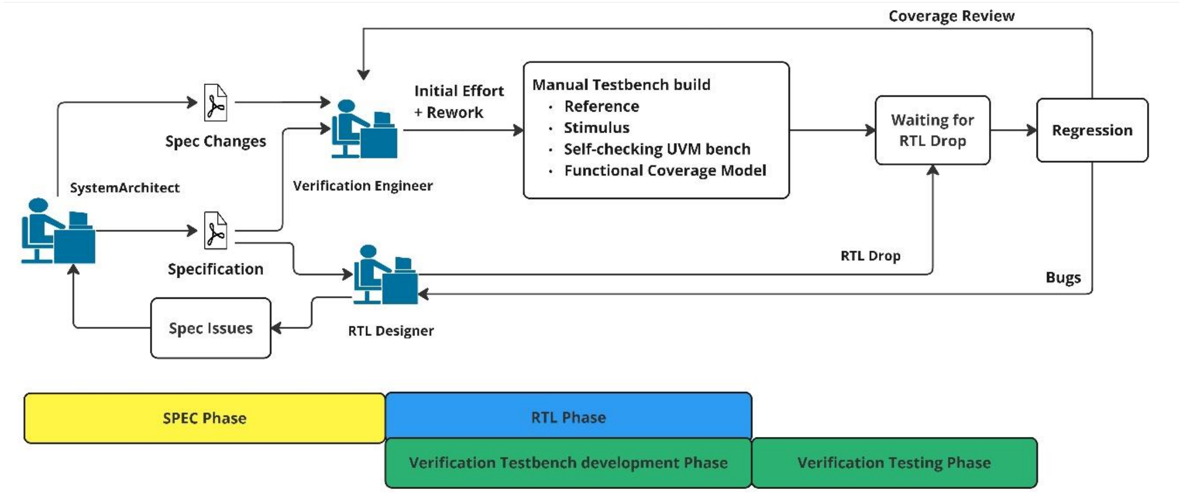


Figure .1 Traditional Verification Workflow

Verification engineers can also leverage these models for accurately capturing algorithm behavior by modeling the verification environment using Simulink and MATLAB. Model-Based Verification is an excellent solution for the Shift-Left verification in terms of a) mapping Simulink tests to requirements in our requirements definition tool Jama for initial verification planning [18], b) being easy to update when requirements change, c) reducing Simulation turnaround time, d) offering easier and faster debugging, e) enabling earlier testing, f) supporting design functionality more quickly, and g) authoring and managing regression test-suites. Furthermore, we can automatically generate standalone UVM components from Simulink that can be integrated into the UVM environment [7, 8].

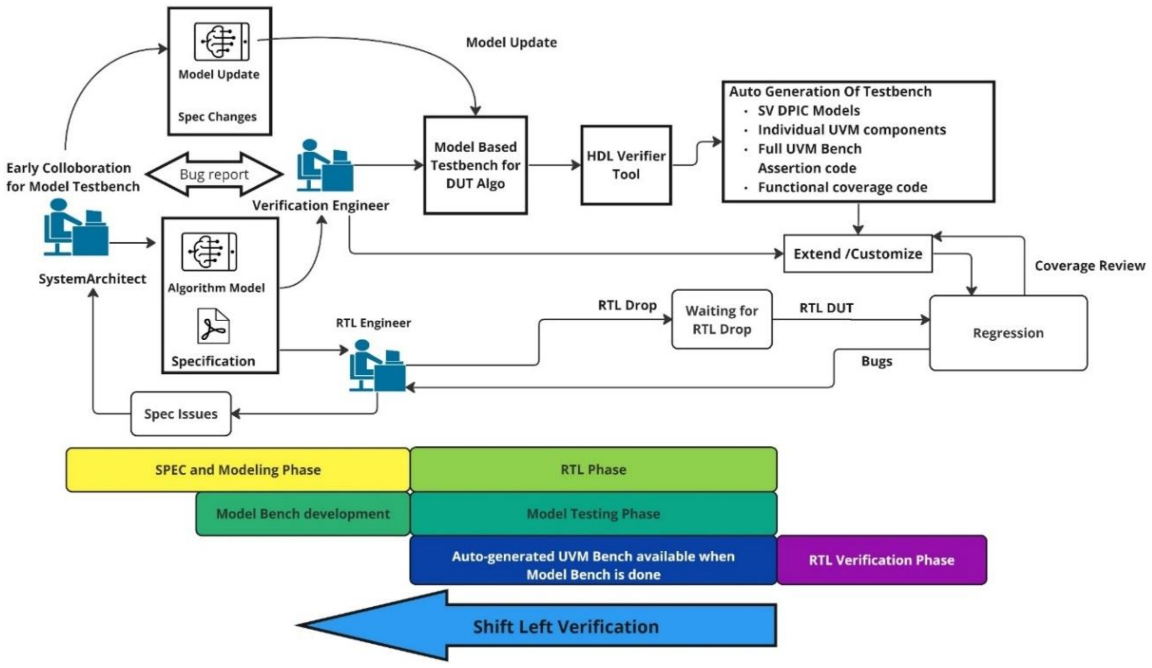


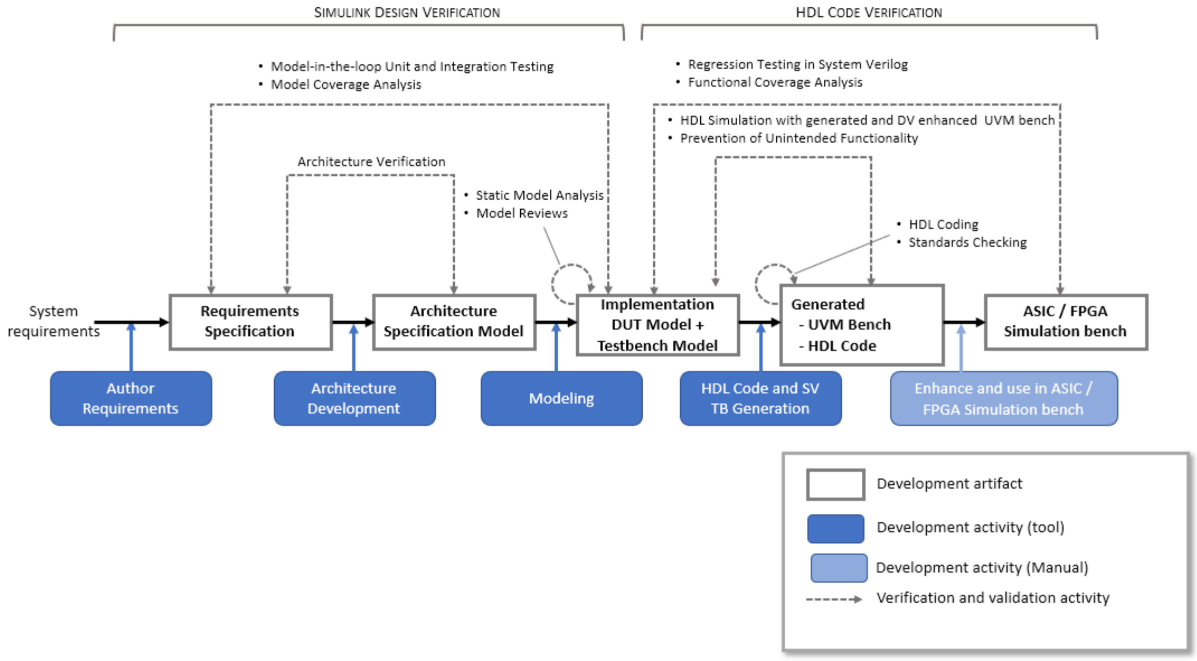
Figure .2 Model-Based Verification.

Model-Based Design starts from capturing requirements and progresses all the way to implementation [9]. As shown in Fig 3, Listed below are descriptions of the key stages in model development workflow.

Requirement Specification - System engineers can initially define requirements and annotate models with these requirements. It helps to ensure traceability of requirements throughout the design cycle.

Architecture Model – System engineers can then build architecture models corresponding to the requirements and simulate the model. They can use this model to fine-tune the algorithms without worrying about implementation details such as converting to fixed-point design and experiment with frame-based versus sample-based architectures.

Implementation Model and Testbench – The next step is to convert architecture models into implementation models. The conversion is needed because fixed point logic is simpler and use less resources on hardware. Not having a floating-point processor to deal with could be another reason. In any case, it is important to select word size and fractions bits such that the range of floating-point results and precision is maintained as per the specification. This conversion may involve a) converting a floating-point algorithm to a fixed-point representation and b) using a subset of Simulink blocks and MATLAB language that support HDL code generation for designs-under-test (DUTs). System Engineers can then collaborate with design verification (DV) engineers to perform Model-based Testing (MBT). Using MBT, the shift-left of verification happens from RTL level to a Model-based DUT. This is achieved by building a Simulink-based testbench and then transitioning into detailed design and verification using models. The models are then verified, and model structural coverage is reviewed. At this stage, formal verification for DUT model can be run to detect dead logic, integer overflow, array access violations and division by zero. The users can also include any legacy RTL IP which has been developed outside model-based workflow into Simulink using HDL co-simulation block [10]. For enabling downstream UVM testbench generation, it is essential that the Stimulus, DUT, and Checker models use MATLAB code and Simulink blocks that support C code generation.



SIMULINK DESIGN VERIFICATION
HDL CODE VERIFICATION
• Model-in-the-loop Unit and Integration Testing
• Model Coverage Analysis
• Regression Testing in System Verilog
• Functional Coverage Analysis
• HDL Simulation with generated and DV enhanced UVM bench
• Prevention of Unintended Functionality
Architecture Verification
• Static Model Analysis
• Model Reviews
• HDL Coding
• Standards Checking
System requirements
Requirements Specification
Architecture Specification Model
Implementation DUT Model + Testbench Model
Generated - UVM Bench - HDL Code
ASIC / FPGA Simulation bench
Author Requirements
Architecture Development
Modeling
HDL Code and SV TB Generation
Enhance and use in ASIC / FPGA Simulation bench
Development artifact
Development activity (tool)
Development activity (Manual)
Verification and validation activity

Figure 3 Model-Based Design and Verification (MBDV)

HDL and UVM Code generation – The regression results and coverage review are used as status of Verification closure at the model level in Simulink. The code generation tools can then be used for HDL code generation from the DUT and UVM testbench generation from testbench components of Simulink model.

RTL Simulation Testbench– The generated block-level UVM bench can then be enhanced by DV engineers and integrated into their top level UVM bench. In this step, functional coverage analysis can act as feedback to update Stimulus model in Simulink to generate missing scenarios.

IV. Example: Model-based testbench

In this section, we discuss how to build a Model-based testbench using a CORDIC design as an example. A CORDIC circuit serves to compute several common mathematical functions, such as trigonometric, hyperbolic, logarithmic, and exponential functions. The key components of a Model Based testbench are shown in figure 4. The description of each of the subsystems is listed below.

1. **Sequence -** This subsystem generates stimulus for the DUT. This block consists of MATLAB function code and other Simulink library blocks to create and randomize stimulus [11, 12]. The sequence block has input called seed, which is used to initialize the MATLAB random number generator [13, 14]. We use a Test Sequence block to create different test scenarios and the selection of which scenario to run is done through an input Simulink parameter [15]. These scenarios are used to switch between a functional testcase and a randomized testcase. Functional testcase is based on the real time data gathered from mechanical setup in the lab which is processed using MATLAB script until the input of CORDIC stage. Randomized testcase utilizes the seed input and MATLAB function code to generate constrained random stimulus. These scenarios can be run from the same Simulink test bench. Simulink Assertions are placed on the fixed-point inputs for range checking. Simulink verify calls are used to collect the coverage on the generated input stimulus [17].

   The Sequence block translates into a uvm_sequence on code generation. This automatic translation is a two-step process. The first step is conversion from a Simulink subsystem into a System Verilog DPI-C model that preserves the behavior as in Simulink. The System Verilog DPI-C model is then further converted into a uvm_sequence. In RTL simulations, the stimulus generation is randomized based on the seed input. If a test fails in RTL simulations, then the seed input value in uvm_sequence for the failing simulation can be plugged into seed input of sequence to recreate the failing scenario in Model-based testbench and debug the design with a Model-Based Design engineer.

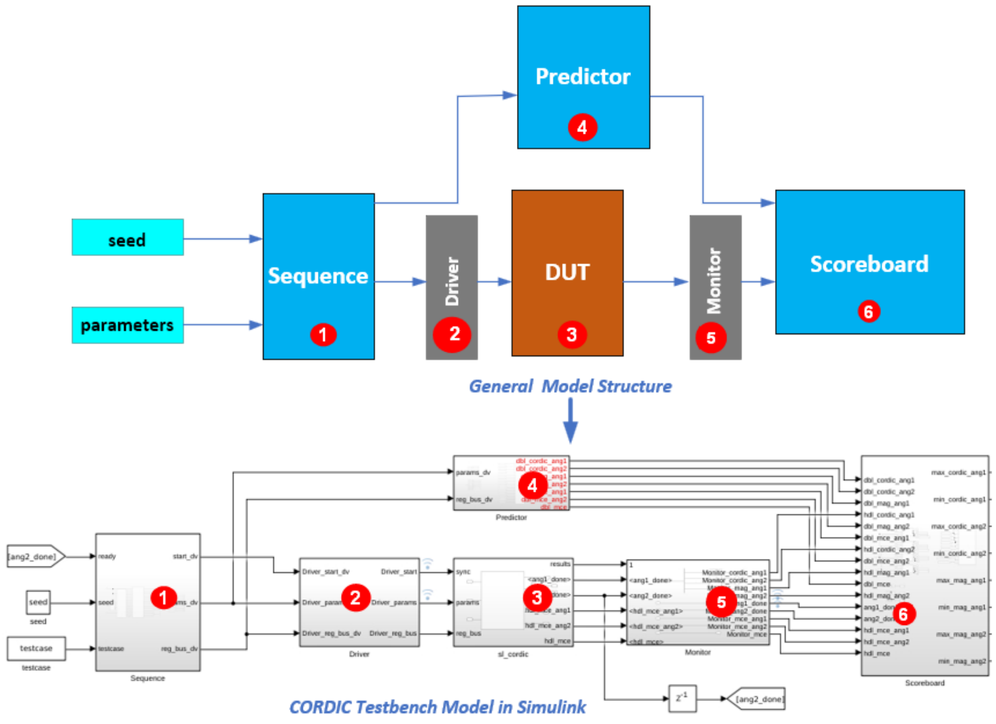
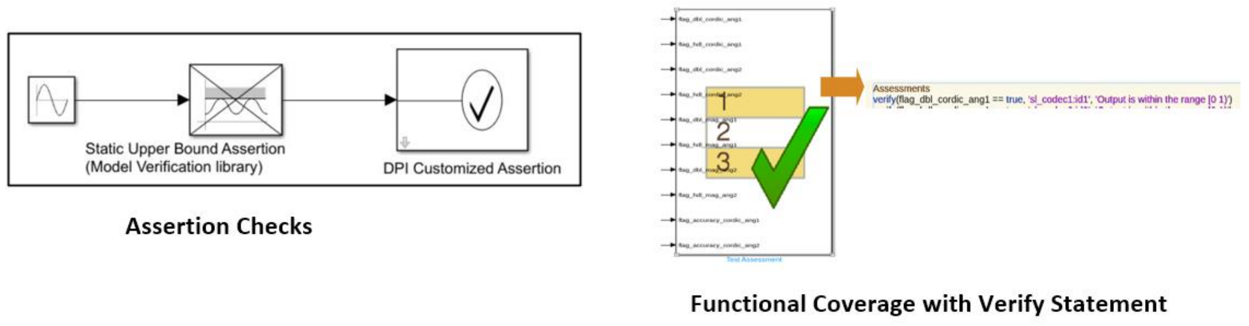


Figure 4 CORDIC Testbench Model

2. **Driver –** This is an optional block and can be used in scenarios where Stimulus is frame-based, generating data as a vector. The DUT model for HDL implementation can accept only scalar input (a sample value at a time), the driver block can then implement the frame to sample conversion. Also, in cases where stimulus is floating point, the driver can be used to do float to fixed point conversion before sending stimulus to DUT which is a fixed point.
3. **DUT**– The DUT is an implementation model of CORDIC algorithm. This model has been developed using Simulink blocks and MATLAB code that supports HDL code generation [16]. The DUT model is a fixed-point model.
4. **Predictor –** The predictor block serves the purpose of reference model being in floating point. It is written at higher level of abstraction without consideration for HDL code generation. The only requirement is the Simulink blocks and MATLAB code used should support code generation to produce SV DPI-C model and UVM component. The predictor block receives input from Sequence block and produces reference output used by scoreboard for self-checking logic. This model can be implemented as floating point model. In the case of CORDIC, the MATLAB code developed here is drawn from the specifications document.
5. **Monitor –** This is also an optional block and can be used in scenarios when output from DUT needs to be converted into a frame for comparison in Scoreboard. The monitor block is used to convert DUT fixed point output to floating point for comparison in scoreboard.
6. **Scoreboard -** The scoreboard subsystem acts as a self-checking block which compares the predictor (ideal model) outputs against the DUT outputs. The DUT fixed point results when converted to floating point should match the predictor model within certain error tolerance. If there is a mismatch above error tolerance, the fixed-point arithmetic in DUT might be incorrect or DUT logic has issues. Assertions are modeled in scoreboard using the 'Assertion for DPI-C' block. Cover-groups are modeled using the verify statement in a Test Sequence or Assessment block [17].



```
verify(flag_dbl_cordic_ang1 == true, 'sl_codec1:id1', 'Output is within the range [0 1)')
```

Figure 5 Assertion checks (left) and functional coverage (right)

UVM BENCH INTEGRATION

The **uvmbuild()** function from MathWorks generates a SystemVerilog top module which includes complete UVM testbench and a behavioral DUT. Each of the subsystems from the model are converted into respective individual UVM components such as uvm_sequence, uvm_driver, uvm_scoreboard [13]. The behavioral DUT is replaced with the RTL generated from the HDL coder while running the simulations.

This generated UVC is integrated into the UVM environment using the following set of steps.

1. Include the generated files in the compilation list.
2. Make the required connections between generated uvmbuild and RTL using uvmbuild interface.
3. Create a new test which reuses the same agent, environment, and sequence. We can further add more constraints, assertions, and cover groups to overcome any limitations in generated uvmbuild code.
4. Inputs will be driven into RTL by triggering the uvmbuild sequences.
5. Data integrity check will be done in uvmbuild scoreboard, where DUT values are compared against expected data from predictor.

DISCUSSION OF BUGS CAUGHT

In the digital signal path of our project, the specification document has equations defined for most of the intermediate blocks. Most of the issues we found while using the MBV flow are related to the definition of equations and data type inconsistencies which is expected since the specification documentation is still in the initial stages of implementation. In the CORDIC block, the bypass and polarity change configurations were added in the later stages of the implementation. When these configurations are enabled, the radius safety flag checks were not being exercised which was identified through MBV. One of the critical issues we found in this project was that the conditional statements were contradictory, and the corresponding logic does not have a valid output for all the input stimulus scenarios. This bug is reported to the Systems Team and the specification document is corrected after reviewing the bug.

LIMITATIONS

The current limitations, which can be listed as future improvements, are listed below:

- Constraints on the input stimulus are limited to the minimum and maximum ranges. Other constraints available in System Verilog such as solve before, weighted distributions are currently not feasible with this flow.
- The inputs stimulus is streamed into the DUT based on the feedback/acknowledge signal received from the DUT. The uvmbuild currently does not support feedback between DUT and Sequence.
- The self-checking bench would need an acknowledge signal from DUT which can be used to synchronize between the predictor output and the DUT output. The predictor computes the output relatively faster than the DUT and the checkers are activated based on the acknowledge signal from DUT.
- Modeling of very complex concurrent assertions is currently a challenge while using the Model Based Verification flow.
- Model Based Verification flow only supports basic cover groups modeling and options.

V. CONCLUSION

With projects that involve developing the design in Simulink tool (Model Based Design) and then exporting RTL code with fixed point, it made sense for Verification teams to get involved with Model Based Verification early on. There was some initial work needed to integrate the generated UVM testbench files into our main chip level UVM environment. The main chip level UVM environment can leverage SVRNM models for Analog blocks (VAMS can be added with some restrictions) at the front end of Digital blocks resulting in streaming in realistic stimulus into the chip and thus verifying the Digital blocks much more efficiently. Modelling at chip-level leveraging SVRNM and UVM-AMS could help connect the dots for audience and show the various possibilities across Digital IP level and Chip level.

Shifting the design verification effort upstream by a verification engineer has resulted in:

- Verification of the models **is more exhaustive early** on because verification engineers are best trained to find out how to break a system.
- Generation of **better-quality RTL** from the models, saving more than two months of verification effort.
- **Reuse of models** with their associated Simulink test environments by **Verification team** for upcoming projects is expected to save two or more months.
- **Reuse of models** by **Systems Engineering** to confirm that the implemented design does what requirements specify.
- An opportunity for Allegro's **customers** to **reuse models** within their own environments to confirm their requirements are met.

Model fidelity can be improved as the design progresses, but bugs can be caught even before the customer has a part in hand. Moreover, we expect the bell curve (Fig.6) on bugs reported coming out of the Model-Based approach to have a lower height (meaning fewer bugs) resulting in fewer regression runs translating to faster verification time and fewer usage of HDL simulator licenses: all contributing to savings on cost of the product.

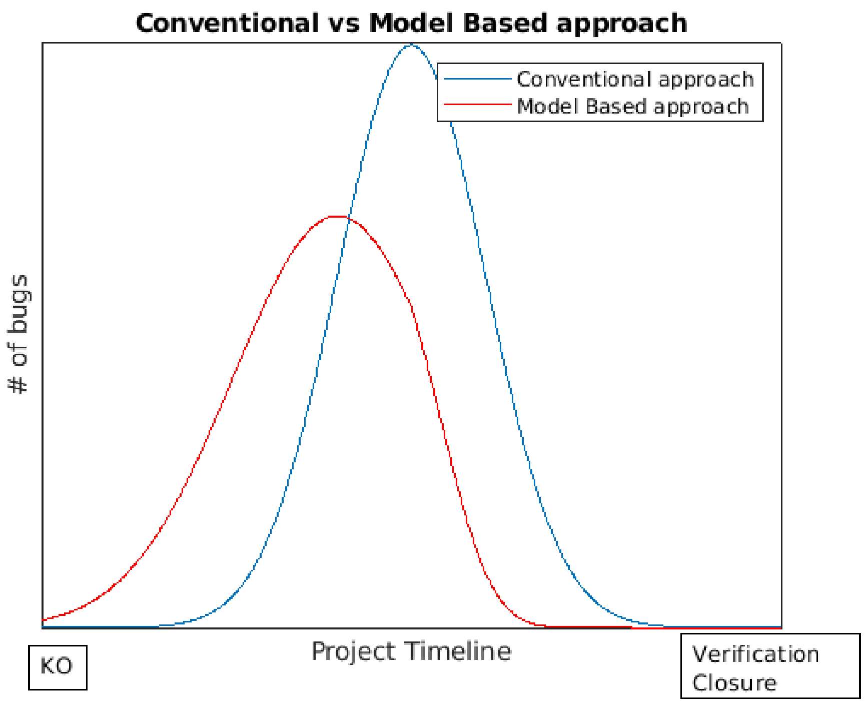


Figure 6 Conventional vs Model Based approach.

VI. References

[1] Jia, Tao and Erickson, Jack. (2015 June). Reuse MATLAB Functions and Simulink Models in UVM Environments with Automatic SystemVerilog DPI Component Generation. Verification Horizons. https://verificationacademy.com/verification-horizons/june-2015-volume-11-issue-2/Reuse-MATLAB-Functions-and-Simulink-Models-in-UVM-Environments-with-Automatic-SystemVerilog-DPI-Component-Generation.

[2] Mentor Graphics Corporation. (2012 February 22). Mentor Graphics Drives Broader Adoption of UVM [Press release]. https://verificationacademy.com/news/mentor-graphics-drives-broader-adoption-uvm.

[3] Baird, Mike and Oden, Bob. (2016 March 1). Slaying the UVM Reuse Dragon: Issues and Strategies for Achieving UVM Reuse [Paper presentation]. 2016 Design and Verification Conference US, San Jose, CA, United States. https://verificationacademy.com/resources/technical-papers/slaying-the-uvm-reuse-dragon

[4] UVMF MathWorks Integration Users Guide, Version 2021.3, retrieved from https://verificationacademy.com/topics/verification-methodology/uvm-framework.

[5] Mentor Graphics. (2017 June 6). Mentor users in New Era of C++ Verification Signoff with New Catapult Tools and Solutions [Press release]. https://www.plm.automation.siemens.com/global/en/our-story/newsroom/mentor-ushers-new-era-c-verification-signoff-new-cataplult/91677.

[6] MathWorks, HDL Verifier documentation, https://www.mathworks.com/help/hdlverifier/uvm-generation.html

[7] Alagna, Diego et al., (2021 October 26-27, 2021), Reuse of System-Level Verification Components within Chip-Level UVM Environments [Paper presentation]. 2021 DVCon Europe, virtual conference, https://dvcon-proceedings.org/wp-content/uploads/reuse-of-system-level-verification-components-within-chip-level-uvm-environments-paper.pdf

[8] Lakshminarayana, Avinash, et al. (2022 February 28 – March 3), Flattening the UVM Learning curve: Automated solutions for DSP filter Verification [Paper presentation]. 2022 DVCon US, virtual conference, https://dvcon-proceedings.org/wp-content/uploads/Flattening-the-UVM-Learning-Curve-Automated-solutions-for-DSP-filter-Verification-1.pdf

[9] Dimitri Hamidi, Dr. Tjorben Gross, Eric Cigan, Tom Richter, “Meeting Functional Safety Standards on Algorithm Implementation for FPGA and ASIC in a Dynamic Automotive Environment”, embedded world 2023. https://www.mathworks.com/company/newsletters/articles/meeting-functional-safety-standards-on-algorithm-implementation-for-fpga-and-asic-in-a-dynamic-automotive-environment.html

[10] HDL Co-simulation with MATLAB or Simulink https://www.mathworks.com/help/hdlverifier/gs/hdl-cosimulation.html

[11] UVM build Reference, https://www.mathworks.com/help/hdlverifier/uvm-generation.html

[12] Generate Parameterized UVM bench from Simulink, https://www.mathworks.com/help/hdlverifier/ug/generate_param_uvm_test_bench_from_simulink.html

[13] MATLAB Random number generation, https://www.mathworks.com/help/matlab/ref/rand.html

[14] Control Random number generator, https://www.mathworks.com/help/matlab/ref/rng.html

[15] Simulink Test Sequence Scenarios, https://www.mathworks.com/help/sltest/ug/define-test-sequence-scenarios-in-test-sequence-editor.html

[16] HDL Code generation Simulink, https://www.mathworks.com/help/hdlcoder/hdl-code-generation-from-simulink.html

[17] Generate system Verilog Assertions and Functional Coverage, https://www.mathworks.com/help/hdlverifier/ug/generate-systemverilog-dpi-from-simulink-tests-verify-statement.html

[18] JAMA connect for Semiconductor, https://www.jamasoftware.com/solutions/semiconductor