ASSERTION BASED VERIFICATION FOR ANALOG AND MIXED-SIGNAL DESIGNS USING SIMULATIONS

by

Doğan Ulus

B.S., Electrical and Electronics Engineering, Boğaziçi University, 2011

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical and Electronics Engineering
Boğaziçi University

2013

ASSERTION BASED VERIFICATION FOR ANALOG AND MIXED-SIGNAL DESIGNS USING SIMULATIONS

APPROVED BY:

Assist. Prof. Faik Başkaya (Thesis Supervisor)	.....
Assoc. Prof. Alper Şen (Thesis Co-supervisor)	.....
Prof. Günhan Dündar	.....
Prof. Can Özturan	.....
Assoc. Prof. Şenol Mutlu	.....

DATE OF APPROVAL: 15.08.2013

ACKNOWLEDGEMENTS

It was the best of times, it was the worst of times. No, well, not that worst actually but I tried a lot and I learned a lot for this thesis. There are many people I would like to express my gratitude so let me start my long list from my advisors. I want to thank Faik Başkaya for his guidance and support starting from my bachelor studies until the finalization of this thesis. The freedom he gave me was extremely valuable, now I understand more. Alper Şen taught me everything I know about formal methods and verification and this thesis would not be existed without his vision but I owe him much more than this, especially for his trust in me, his support and various fruitful discussions.

I would like to thank Günhan Dündar for his advices at various stages of this thesis. BETA laboratory is the place where I was spattered with analog design therefore İsmail, Barış, Okan, Melih, Baykal, Engin, Berk and all other members need a special thanks for this great working atmosphere. Besides BETA members, I thank Abdullah, İsmail, Berkan, Gökhan, Uraz for their friendship and good times we spent together. Also my long-long time friends, Barış, Aybars, Gizem, Aybike, Peyvent and not-that-long-but-strong friends Melih, Yasin, Oktay, and Efe boosted my morale at anytime I need a refreshment.

I thank to Scott Little for his initial advices, to Dejan Nickovic and to Oded Maler for sending source code of their Analog Monitoring Tool.

Finally, I would like to thank my parents, Ulviye and Merih Ulus for their continuous support.

ABSTRACT

ASSERTION BASED VERIFICATION FOR ANALOG AND MIXED-SIGNAL DESIGNS USING SIMULATIONS

This thesis studies assertion based verification methodology for analog and mixed-signal (AMS) designs and improves analog expressiveness of assertions. Assertion based verification methodology is originally derived for digital domain, hence AMS assertion languages are inadequate to express all aspects of AMS designs. Therefore, we first introduce the halo concept for analog signals to formally express them with their tolerance and variation values in assertions. Haloes of analog signals define an effective region around these signals which help analog comparison. Second, we integrate measurements and circuit analyses into AMS assertions. These analyses are widely used verification techniques in conventional AMS verification. Their integration into assertions provide a complete and unified AMS verification methodology. Finally, we develop AMS-Verify, a flexible framework to verify AMS properties on simulations. AMS-Verify is able to express analog tolerances, measurements and circuit analyses. We validate our solutions in three case studies using AMS-Verify framework.

ÖZET

ANALOG VE KARIŞIK İŞARET TASARIMLARIN BENZETİMLER ÜZERİNDEN GÖZCÜ TABANLI DOĞRULAMASI

Bu tezde Analog ve Karışık İşaret (AKİ) sistemler için gözcü tabanlı doğrulama yöntembilimi incelenmiş ve gözcülerin analog özellikleri daha ifade edebeilmesi sağlanmıştır. Gözcü tabanlı doğrulama yöntemleri, ilk olarak sayısal tasarım ve yazılım alanları için geliştirildiği için analog özellikleri ifade etmekte yetersiz kalmaktadır. Bu yüzden, öncelikle analog işaretler için hale kavramını öne sürülmüştür. Haleler, analog işaretlerin çevresinde kabul edilebilir alanlar tanımlayarak analog karşılaştırmayı kolaylaştırmaktadır. İkinci olarak analog doğrulamada yaygın olarak kullanılan ölçümler ve devre çözümlemeleri AKİ gözcüleri ile bütünleştirilmiştir. Bu bütünleşme ile AKİ sistemler için tam ve birleşik bir doğrulama ortamı elde edilmiştir. Son olarak AMS-Verify aracı geliştirilmiş ve önerilen doğrulama çözümleri bu araç ile gerçeklenmiştir. Sunulan üç örnek tasarımda ise AKİ özellikler için gözcü formülleri yazılmış ve bu özellikler AMS-Verify aracı kullanılarak benzetimler üzerinden doğrulanmıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . xi
LIST OF SYMBOLS . . . xii
LIST OF ACRONYMS/ABBREVIATIONS . . . xiii
1. INTRODUCTION . . . 1
1.1. AMS Design and Verification . . . 3
1.2. Formal Verification . . . 4
1.3. Related Work . . . 5
1.4. Contributions . . . 6
1.5. Publications . . . 7
1.6. Thesis Organization . . . 7
2. AMS ASSERTIONS . . . 8
2.1. What is an Assertion? . . . 8
2.2. Temporal Logic . . . 9
2.3. AMS Assertion Languages . . . 11
3. ANALOG TOLERANCES AND COMPARISON . . . 14
3.1. Halo Concept . . . 14
3.2. Halo Calculation . . . 17
3.2.1. Absolute Method . . . 17
3.2.2. Relative Method . . . 17
3.2.3. Smoothing Method . . . 18
3.2.4. Monte Carlo Method . . . 20
3.3. Analog Comparison . . . 20
3.3.1. Analog Signal Representation . . . 20
3.3.2. Booleanization . . . 21

4. INTEGRATING MEASUREMENTS AND CIRCUIT ANALYSES 24
4.1. Event Representation 24
4.2. Measurements for AMS Assertions 26
4.2.1. General Measurements 26
4.2.2. Time Domain Measurements 27
4.2.3. Frequency Domain Measurements 28
4.3. Circuit Analyses 29
4.3.1. DC Operating Point Analysis and DC Assertions 29
4.3.2. AC Analysis and AC Assertions 31
4.3.3. Fourier Analysis for Noise and Linearity 34
5. AMS-VERIFY FRAMEWORK 39
5.1. Input File Structure 41
5.2. Assertion Evaluation 42
6. CASE STUDIES 44
6.1. Settling Time Property 44
6.2. SH and DAC 48
6.3. Programmable Gm-C Filter 51
7. CONCLUSION 58
APPENDIX A: AMS-VERIFY TUTORIAL 60
A.1. Writing Assertion Suites 60
A.2. Running Assertion Suites 62
APPENDIX B: ASSERTION SOURCE FILES 63
REFERENCES 70

LIST OF FIGURES

Figure 2.1. Overview of an assertion-based verification flow. 8

Figure 2.2. Boolean proposition p and evaluation of $\mathcal{F}_{[1:2]}\ p$. 10

Figure 3.1. Haloes show tolerances for analog signals. 15

Figure 3.2. Comparison with tolerances. 16

Figure 3.3. Halo calculated by an absolute tolerance value, which is 0.05V. . . 18

Figure 3.4. Halo calculated by a relative tolerance value, which is is 5%. . . . 19

Figure 3.5. Halo calculated by smoothing reference signal. Tolerance value is 0.05V. 20

Figure 4.1. Events can represent measurement results for signals. 25

Figure 4.2. Events can be translated into Boolean and analog signals. 25

Figure 4.3. An example AC analysis plot and measured AC specifications. . . 32

Figure 4.4. Monitoring time-varying AC characteristics in the middle of time-domain circuit simulation. 33

Figure 4.5. Power Spectrum Example. 34

Figure 4.6. THD Example. 36

Figure 4.7. SFDR Example. 37

Figure 5.1. Overview of AMS-Verify Framework. 39

Figure 5.2. Example input file. 40

Figure 5.3. Signals for assertion evaluation of input file in Figure 5.2. 42

Figure 5.4. Output of assertion evaluation of input file in Figure 5.2. 43

Figure 6.1. Evaluation results of subformula r and f for the signal a. 46

Figure 6.2. Evaluation results of subformula r and f for the signal b. 47

Figure 6.3. Output signals of SH and DAC circuit as well as triggering clock signal. 49

Figure 6.4. Evaluation steps of Assertion 6.9. 50

Figure 6.5. The programmable low-pass filter circuit. 51

Figure 6.6. Monitoring time-varying DC level of the filter output according to K. 52

Figure 6.7. Monitoring time-varying cutoff frequency value of the filter according to K factor. 54

Figure 6.8. Bandwidth vs K and fitted line. 55

Figure 6.9. Monitoring time-varying THD and SNDR values of the filter according to K. 57

Figure B.1. Source file of Settling Time Property in Assertion 6.8. 63

Figure B.2. Source file of SH and DAC property in Assertion 6.9. 64

Figure B.3. Source file of DC voltage property of programmable filter in Assertion 6.10. 65

Figure B.4. Source file of AC voltage property of programmable filter in Assertion 6.11. 66

Figure B.5. Source file of regression property of programmable filter in Assertion 6.12. 67

Figure B.6. Source file of THD property of programmable filter in Assertion 6.13. 68

Figure B.7. Source file of SNDR property of programmable filter in Assertion 6.14. 69

LIST OF TABLES

Table 1.1. Comparison of assertion based verification methodology. 2

Table 2.1. Our AMS Assertion Language Grammar. 13

Table 3.1. Computation a boolean signal from sample values. 22

LIST OF SYMBOLS

$\mathcal{F}$	Temporal Eventually Operator
$\mathcal{G}$	Temporal Always Operator

LIST OF ACRONYMS/ABBREVIATIONS

AC	Alternating Current
ADC	Analog to Digital Converter
AMS	Analog and Mixed Signal
CTL	Computation Tree Logic
DAC	Digital to Analog Converter
DC	Direct Current
FFT	Fast Fourier Transform
LPF	Low-Pass Filter
LTI	Linear Time-Invariant
LTL	Linear Temporal Logic
MITL	Metric Interval Temporal Logic
MTL	Metric Temporal Logic
PGA	Programmable Gain Amplifier
PORV	Predicates Over Real Variables
PSL	Property Specification Language
RC	Resistor-Capacitor
SAT	Satisfiability Problem
SFDR	Spurious Free Dynamic Range
SNDR	Signal and Distortion Ratio
SH	Sample and Hold
SPICE	Simulation Program with Integrated Circuit Emphasis
THD	Total Harmonic Distortion

1. INTRODUCTION

Analog and mixed-signal (AMS) electronics community is challenged by ever increasing system complexity and decreased time to market in past years. From ambitious consumer gadgets to safety-critical medical applications, virtually every electronics device now uses components containing various AMS modules. However, AMS design process is more vulnerable to faults more than ever because AMS design flows are insufficient to handle requirements such as high performance, low power and reliability in a short time. Therefore, verification of AMS modules becomes the biggest bottleneck in the development cycle of high-end electronic systems.

When digital electronics and software community was faced with a similar complexity increase problem, formalizing design and verification flows helped to cope with this problem. As a solution, formal methods such as model and equivalence checking have been proposed and studied extensively to verify designs to get a full verification coverage in recent decades. However, although formal methods ensure 100% correctness of designs, these methods suffer from the state-explosion problem. As an alternative, lightweight semi-formal approaches have been gaining popularity in the industry thanks to their simplicity and scalability. These semi-formal approaches are called assertion-based verification, runtime verification or monitoring but all these methods are doing the same things: they observe the system and check the given formal specification on the simulations. Although semi-formal approaches are incomplete and do not guarantee full system correctness, these approaches are effectively used to catch unintended behaviors in the system. In Table 1.1, we compare assertion-based verification with formal and conventional verification in terms of base objects, definition of specifications, verification coverage, easiness of setup, evaluation of specifications as well as intuitiveness, structuredness and scalability of verification process.

Similarly, analog and mixed-signal community have mainly relied on simulation-based verification and post-simulation analysis techniques. These ad-hoc solutions do not usually include a support for automation and reusability, and require considerable

Table 1.1. Comparison of assertion based verification methodology.

	Conventional	ABV	Formal
Base	Simulations	Simulations	Models
Specs	Informal	Formal	Formal
Coverage	Limited	Limited	Full
Setup	Medium	Medium	Hard
Intuitive	Yes	Yes	No
Structured	No	Yes	Yes
Scalable	Yes	Yes	No
Evaluation	Manual	Auto	Auto

amount of time and user effort. This situation is even worse when same AMS properties should be checked from simulation results at every level of abstraction from algorithmic specification to actual implementation. Different abstraction levels require different tools and different specification formats. This means that designers have to spend their valuable time to translate specifications for different tools, perform simulations and evaluate results repeatedly. As a solution, formalizing AMS verification practices in a unified environment increases productivity and reliability of modern AMS design as well as reduces development time.

The general motivation of this thesis is to study semi-formal assertion-based verification methodology for AMS designs. We check behaviors of AMS systems with respect to a high-level specification written in our AMS assertion language. We extend some early works on AMS assertion languages with useful constructs of conventional analog verification, and we implement all these constructs in a flexible framework. We first explain analog and mixed-signal design and verification in Section 1.1 in more detail. We briefly introduce formal verification in Section 1.2. These sections form the background of this thesis and we combine elements from both of these vast research areas in the context of this thesis. In Section 1.3, we give an overview of assertion based AMS verification from the literature. Section 1.4 summarizes our contributions in this thesis and Section 1.5 shows publications that emerged from this research work. We finally conclude the introduction with Section 1.6 by describing the thesis organization.

1.1. AMS Design and Verification

Analog design flow starts in the behavioral domain with a concept and ends in the physical domain with a layout on silicon. Between these steps, the concept should be hierarchically translated into various representations from algorithmic implementation to circuit and device implementations. Every translation between abstraction levels brings a growing number of irregularities but it is a significant challenge because precision and accuracy are very important aspects of analog design. An analog designer should be aware of these irregularities and handle them accordingly for a reliable operation. For example, every new technology node increases process variations and aging effects in analog circuits; therefore, many current research and industry efforts have been made to develop more robust analog and mixed-signal systems by devising digitally-assisted or calibrated analog circuits.

Such challenges and new paradigms in AMS design increases design complexity, makes existing AMS verification methodologies insufficient. Traditionally, numerical simulation is the core of AMS verification and it's backed with a number of post-simulation analysis techniques such as Fourier analysis, histograms and eye diagrams. Waveform calculators, measurement commands and manually written scripts are ad-hoc solutions to apply these techniques in AMS verification environments but tool support to automatize and reuse these solutions across designs require considerable amount of time and user effort.

To employ a structured AMS verification methodology, we look at the sources of errors in a typical AMS design. AMS verification aims to find three types of errors:

(i) Errors within analog blocks. For example, wrong bias values for amplifiers or cutoff frequency for filters fall into this category. These errors can be detected by traditional analog verification but designer often ensures correctness for a few test cases and assumes that the design is correct for other cases, which may not always be true.

(ii) Errors between analog blocks. For example, inside a PLL design a voltage-

controlled oscillator and low-pass filter should operate together, and errors on the interface of these blocks may be unnoticed until co-simulation of these blocks. These kinds of systems are initially simulated at a higher level of abstraction before a circuit-level simulation. Therefore, verification techniques should be the same (at least compatible) between these levels. This is the point where traditional analog verification begins to fall short.

(iii) Errors between analog blocks and digital blocks. This type of errors are the hardest to detect because analog and digital domains have different natures.

1.2. Formal Verification

Nowadays our daily lives depend on the reliable operation of the electronic systems. Faults in electronic systems may cause severe consequences ranging from economic catastrophes to loss of human lives. Therefore, formal verification, proving the correctness of a system with respect to some formal specification, is an important study in both industry and academia.

There are two major approaches to formal verification: model checking (MC) and equivalence checking (EC). Model checking proves that the "model" of the electronic system is correct for every possible input and internal state of the system with respect to a property specification in a formal specification language. If the model does not satisfy the property, all violations are detected and counterexamples are returned. Equivalence checking proves the functional equality of two implementations of a design. The implementations can be of different abstraction levels and different description methods such as transistor netlists and hardware description languages.

In current setting, formal verification tools are established in the digital domain while industrial analog circuit design flows are lacking formal or semi-formal verification methodologies. However, formal verification has flaws as well: verifying large systems exhaustively is not feasible and there is a need to create a detailed model of the system, which is usually a very involving task. On the other hand, semi-formal verification does not need a model of the system (instead the system is simulated). It validates

only some of properties of the system and can be applied over very large systems.

1.3. Related Work

Verifying specifications of analog and mixed-signal designs involves analysis of continuous and hybrid systems. A lot of effort has been invested in techniques to analyze the characteristics of these systems in a formal and systematic way. Efforts to formalize analog and mixed-signal verification are divided as formal approaches and simulation based semi-formal approaches. Basics and overview of these formal and semi-formal approaches are presented in [1–3].

One formal approach is to perform model checking over approximated discretized models converted from continuous dynamics represented by differential equations. First in [4], and later in [5–7] model checking algorithms are studied for nonlinear analog circuits. In [8], recurrence equations are used to model analog circuits symbolically as well as labeled hybrid petri-nets are used to generate models from simulation traces as in [9]. Recent directions for analog model checking involve probabilistic approaches in [10, 11] and satisfiability (SAT) solvers in [12, 13]. Reachability analysis is another formal approach exploring state-space to check if there is some "bad" states in all reachable states from initial states. In [14–16], reachability analysis for hybrid systems is studied in detail. Another formal approach equivalence checking ensures that the same input-output behavior is satisfied for different implementations of designs and it is investigated for analog circuits in [17–19].

On the other hand, the numberof state variables and the complexity of formal methods increases exponentially by thedesign size and the application of formal methods to large designs becomes infeasible. Alternatively, lightweight assertion based approaches use simulation results with formal property specifications. Expressive monitors and checkers are crucial for simulation-based approach, therefore several specification languages have been proposed to describe assertions addressing different aspects of AMS verification. In most monitoring techniques, predicates over real-valued signals convert real-valued signals into Boolean signals and temporal properties are

monitored and checked in finite simulation traces. In [20–23], monitoring algorithms using temporal logic is proposed to verify properties of continuous signals. In [24], AMS assertions are implemented as both an extension of SystemVerilog Assertions (SVA) and Open Verification Library (OVL), which are commercial verification platforms for digital systems. Mixed Signal Assertion Language (MSAL) in [25] and Analog Specification Language in XML (ASDeX) in [26] are proposed to express AMS properties formally. In [27, 28], Analog Monitoring Tool (AMT) and Signal Temporal Logic (STL) framework are presented to monitor time-domain properties of continuous and hybrid systems. In [29], AMS-LTL^L is presented as an extension of STL framework with state machines and local variables. In [30], time-frequency analysis is integrated into STL framework.

1.4. Contributions

This thesis is motivated by the need to increase analog expressiveness of existing AMS assertion languages. As assertion-based verification methodology has originally been exported from digital verification, early AMS languages have limited expressiveness for analog facts, which spread over a wide spectrum. Therefore, we believe enriching AMS assertions abilities with analog-oriented constructs would be beneficial for AMS verification, and it would be another step forward for more structured and systematic verification of AMS designs.

(i) We propose an expressive analog layer for AMS assertion languages. One aspect of our proposal is to improve analog comparison and booleanization for handling tolerances in analog designs. Another aspect is to seamlessly integrate events, measurements and circuit analyses into assertion based verification flow.

(ii) We developed AMS-Verify, an AMS verification framework to implement our ideas. Our framework has many features such as signal definitions, managing multiple assertions, temporal, boolean and analog operations over signals as well as interfacing waveform calculators. We implemented our framework in Python language and it has several advantages such as using mature scientific libraries

directly in assertions.

(iii) We applied assertion based verification methodology to three case studies using AMS-Verify framework.

1.5. Publications

Parts of this thesis have been published in [31–34]. Case studies for AMS designs studied in [31]. Tolerances are usually associated with analog signals and the need for supporting tolerances in AMS assertions is investigated in [32]. Useful measurement routines and dual-threshold Booleanization are proposed in [33] as useful extensions for AMS assertion languages to address low-level analog properties. Integration of circuit analyses, which have crucial importance in analog verification, into assertion-based AMS verification flow is presented in [34].

1.6. Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, we give the definiton of assertions and explain general characteristics and the structure of assertion languages for AMS designs. Chapter 3 discusses analog tolerances via halo concept, halo calculation methods and analog comparison which is essential to convert analog signals into Boolean signals in a natural way. In Chapter 4, our event representation is given. Measurements and circuits analyses performed when events occur is explained in detail. In Chapter 5, our assertion based verification framework is introduced and our input file structure and evaluation of properties are described. Three case studies are investigated in Chapter 6 demonstate our ideas and our verification framework. In Chapter 7, conclusions and future work directions are given.

2. AMS ASSERTIONS

2.1. What is an Assertion?

An assertion is defined as a statement about design specification or property that is expected to hold for a design. Assertions are generally used to monitor bad behavior of designs and a failing assertion indicates an error or undesired condition in the design. In some cases, counter-examples can be generated after failed assertions. Counter-examples show why the assertions are failing step-by-step, which makes debugging easier and manageable. Such benefits make assertions valuable instruments for verification tasks.

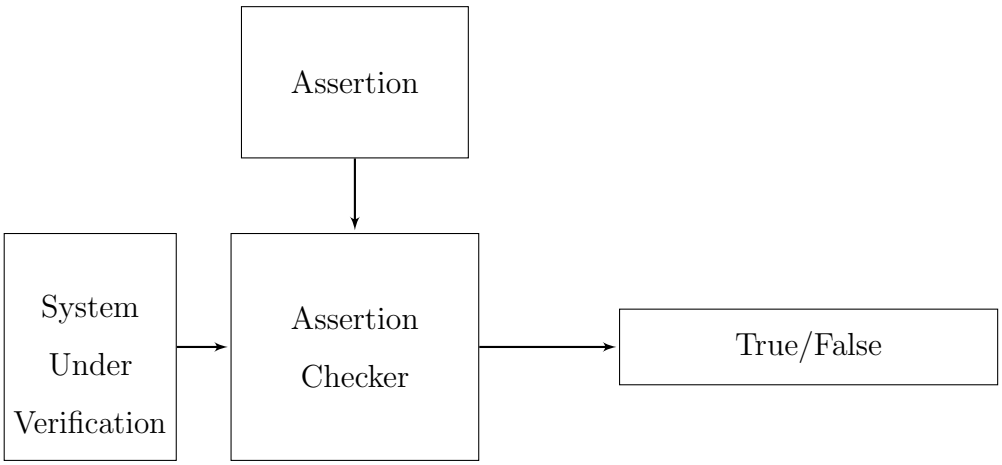


Figure 2.1. Overview of an assertion-based verification flow.

Historically, designers have used assertions to support other verification methods. However, assertions are extended and utilized by employing a methodology as a main tool for various verification duties. Assertion based verification methodology can be defined as verifying design properties using assertions over simulation traces. For assertion based verification methodology, assertions are written in a formal assertion language to capture design specification in a concise and unambiguous manner. In Figure 2.1, we show an overview of assertion-based verification flow. System under verification is checked by an assertion checker according to assertions written in the assertion language. Then, the result is true if checked assertion is satisfied else the

result is false and a counter example is generated.

Although assertion based methodology has originally been developed for software and digital hardware systems, this methodology is slowly expanding towards analog and mixed-signal (AMS) design. In the context of this thesis, we investigate AMS assertions to verify properties of AMS designs. We explain characteristics of AMS assertion languages in Section 2.3.

2.2. Temporal Logic

Assertion based approaches often rely on temporal logic specifications. Temporal logic is an extension of Boolean logic allowing us to reason about the truth of Boolean propositions in terms of *time*. For example, truth value of a proposition such as "I'm in Istanbul." can vary depending on time although the proposition itself does not change. Similarly, an AMS circuit can satisfy specifications for some time instants but violate thereafter. To reason about time and detect violations in time, assertion languages use temporal logic as core language object.

Linear Temporal Logic (LTL) [35] and Computation Tree Logic (CTL) [36] are two main branches of temporal logic, where differ in the perception of time concept. LTL understands the time as a linear object, while CTL perceives the time as a branching, tree-like object. Both logics have very important application areas in the context of verification, and both can be used for AMS verification. In this thesis, linear time notion is naturally suitable for our purposes because we work with simulation traces and these are linear by nature.

LTL, which is first proposed to verify temporal properties of software programs, is extended to have continuous time semantics in Metric Temporal Logic (MTL) [37] and in Metric Interval Temporal Logic (MITL) [38]. Later, in Signal Temporal Logic (STL) [39], MITL is extended to reason about analog signals using predicates over real variables (PORV). For this thesis, we consider a variant of the STL logic with future temporal operators. The formulas are built from propositions using Boolean

connectives and time-bounded version of the *eventually* operator. The syntax of defined by the grammar:

$$\varphi \ := \ p \mid \neg p \mid \varphi_1 \ \vee \ \varphi_2 \mid \ \mathcal{F}_I \ p$$

where p belongs to a set $P = \{ \ p_1, \ldots, p_n\}$ of propositions and I is a nonsingular interval of the form $[a : b]$ where $0 \leq a < b$ are rational numbers. As in STL, the basic operators can be used to derive other standard Boolean and temporal operators, particularly *always* operator:

$$\mathcal{G}_I \ p \equiv \neg(\mathcal{F}_I \ \neg p)$$

For the temporal logic formula φ and a Boolean signal ω, the satisfaction relation $(\omega, t) \vDash \varphi$ indicating that the signal ω satisfies φ at time t is defined as follows:

$$\begin{array}{rll}
(\omega, t) \vDash p & \text{iff} & p[t] \vDash \top \\
(\omega, t) \vDash \neg p & \text{iff} & (\omega, t) \nvDash p \\
(\omega, t) \vDash \varphi_1 \vee \varphi_2 & \text{iff} & (\omega, t) \vDash \varphi_1 \text{ or } (\omega, t) \vDash \varphi_2 \\
(\omega, t) \vDash \mathcal{F}_{[a:b]} \ \varphi & \text{iff} & \text{for some } t' \in \ [t + a : t + b], (\omega, t') \vDash \varphi
\end{array}$$

Note that a formula φ is satisfied by ω if $(\omega, 0) \ \vDash \ \varphi$.

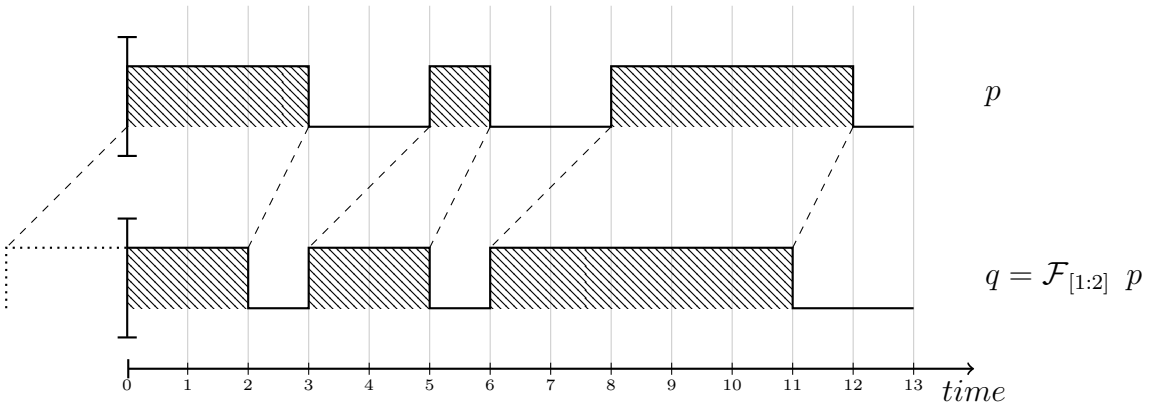


Figure 2.2. Boolean proposition p and evaluation of $\mathcal{F}_{[1:2]} \ p$.

In Figure 2.2, we show a Boolean proposition p and evaluation of the bounded *Eventually* operation, which is $\mathcal{F}_{[1:2]}\ p$. The operator $\mathcal{F}_{[1:2]}$ evaluates true for time instants if p is true within 1 and 2 time units, and returns q. For example, q is evaluated as false at $t = 2.5$ because p is not eventually true in the interval $[3.5, 4.5]$. Similarly, q is evaluated as true at $t = 3.5$ because p is eventually true in the interval $[4.5, 5.5]$.

2.3. AMS Assertion Languages

Analog mixed-signal (AMS) assertion languages consist of several abstraction layers and are influenced by *Property Specification Language* (PSL) [40]. Alongside *Boolean* and *Temporal* layers as in PSL, AMS assertion languages introduce an *Analog* layer to capture analog specifications. In early AMS assertion languages, analog layer is only meant to apply predicates over (real valued) analog signals. However, analog tolerances, events, measurements and circuit analyses used in traditional analog verification are crucial to verify all aspects of analog design. Therefore, we extend our AMS assertion language to handle analog tolerances via haloes, events such as crossing events, measurements such as risetime measurements, and circuit analyses such as DC, AC and Fourier analyses.

We show the grammar of our AMS assertion language in Table 2.1. The temporal and boolean operators have their semantics explained in Section 2.2. Comparison operators as well as *inrange* and *compare* methods are predicates to convert analog signals into Boolean signals. Event operators $\mathcal{E}^a$ and $\mathcal{E}^d$ detect events from analog and digital signals. These operators include *crossing* and *intersect* operators for analog signals as well as *rise* and *fall* operators for digital signals. Measurement operators $\mathcal{M}$ perform corresponding measurement operations over analog signals. *datatowf* operator is to define custom signal from given time-value sequences. *datatowf* operator is useful to define reference signals. DC, AC, $TRAN$ and FFT operators denote corresponding circuit analysis operators. AC property operators, shown as $\mathcal{A}$, calculate AC properties like bandwidth BW from a given AC analysis. Similarly, FFT property operators, shown as $\mathcal{F}$, calculate FFT properties like total harmonic distortion THD from a given FFT analysis. An example assertion using our grammar is as follows:

Assertion

$$Assert: \qquad \mathcal{G}\left(risetime(\ Vout\ )@\big(rise(clk)\big) < 12ns\right) \tag{2.1}$$

end

where $Vout$ denotes an output node of an analog filter. In this assertion, whenever an rising event on the digital signal *clk* occurs, *risetime* operator in the analog layer calculates the risetime value. Then, the risetime value is checked to see whether it is less than 12ns in the Boolean layer, and the $\mathcal{G}$ operator in Temporal layer checks whether this fact is true for all time instants in the simulation.

Table 2.1. Our AMS Assertion Language Grammar.

Temporal Layer	$TempExpr$	::=	$\circledcirc BoolExpr$
		|	$\circledcirc TempExpr$
		|	$TempExpr \bullet TempExpr$
		|	$\neg TempExpr$
Boolean Layer	$BoolExpr$	::=	$SignalExpr \boxtimes SignalExpr$
		|	$inrange(SignalExpr)$
		|	$compare(SignalExpr, SignalExpr)$
		|	$BoolExpr \bullet BoolExpr$
		|	$\neg BoolExpr$
Analog Layer	$SignalExpr$	::=	$RawSignal \odot RawSignal$
		|	$RawSignal$
		|	$Const$
	$RawSignal$	::=	$dcExpr$
		|	$acExpr$
		|	$tranExpr$
		|	$fftExpr$
		|	$measExpr$
		|	$dataExpr$
	$dcExpr$	::=	(Node)@DC($EventsExpr$)
	$acExpr$	::=	$\mathcal{A}$(Node)@AC($EventsExpr$)
	$tranExpr$	::=	Node
	$fftExpr$	::=	$\mathcal{F}(SignalExpr)$@FFT($EventsExpr$)
	$measExpr$	::=	$\mathcal{M}(SignalExpr)$@($EventsExpr$)
	$dataExpr$	::=	$datatowf$(data)
	$EventsExpr$	::=	$\mathcal{E}^a(SignalExpr)$
		|	$\mathcal{E}^d(BoolExpr)$

Legends

$\circledcirc$ Temporal Operators	DC	DC Analysis Operator
$\bullet$ Binary Boolean Operators	AC	AC Analysis Operator
$\neg$ Boolean Negation Operator	FFT	Fourier Analysis Operator
$\odot$ Arithmetic Operators	$\mathcal{A}$	AC Property Operators
$\boxtimes$ Comparison Operators	$\mathcal{F}$	Fourier Property Operators
$\mathcal{E}^a$ Analog Event Operators	$\mathcal{M}$	Measurement Operators
$\mathcal{E}^d$ Digital Event Operators		

3. ANALOG TOLERANCES AND COMPARISON

Analog design is always associated with tolerances by nature. Unlike digital design, you seek for "good-enough" or "close-enough" solutions rather than exact equality for nominal design specifications. System complexities and process variations create deviations from design specifications. Deciding what an acceptable deviation from desired value is often remain to designers' experience in traditional analog design flow. Therefore, it may be a source of ambiguity at the interface of different analog blocks especially when these analog blocks are designed by different designers. Specifying tolerance values formally reduces such errors and makes automatic evaluation of simulation results more natural and realistic.

3.1. Halo Concept

In astronomy a halo is defined as a circular band of colored light visible around the sun or the moon. As a naive interpretation, a halo is a kind of boundary for the object at the center that determines its effective region. Translating this notion into the analog signal domain, we define a pair of boundaries (upper and lower) for each sample of the analog signal we're interested in and name this as the Signal Halo or just the Halo, in short. Further, the area covered by these boundaries in two-dimensional space of analog domain is called the Halo Region. To visualize the halo concept, an arbitrary analog signal and its halo are plotted in Figure 3.1. The methods to calculate such a halo for an analog signal are explained in Section 3.2.

A halo defines a tolerated region around the reference analog signal; therefore, provides a space to relax analog comparisons and allows to assert more realistic properties by considering tolerances in specifications. As a common task in analog verification, two analog signals (eg. a test signal and a golden reference signal) should be compared for equivalence. Although comparing two analog signals is an intuitive task for human eye, automating analog comparisons is difficult. Calculating tolerance values would be the single most important piece of analog comparison techniques.

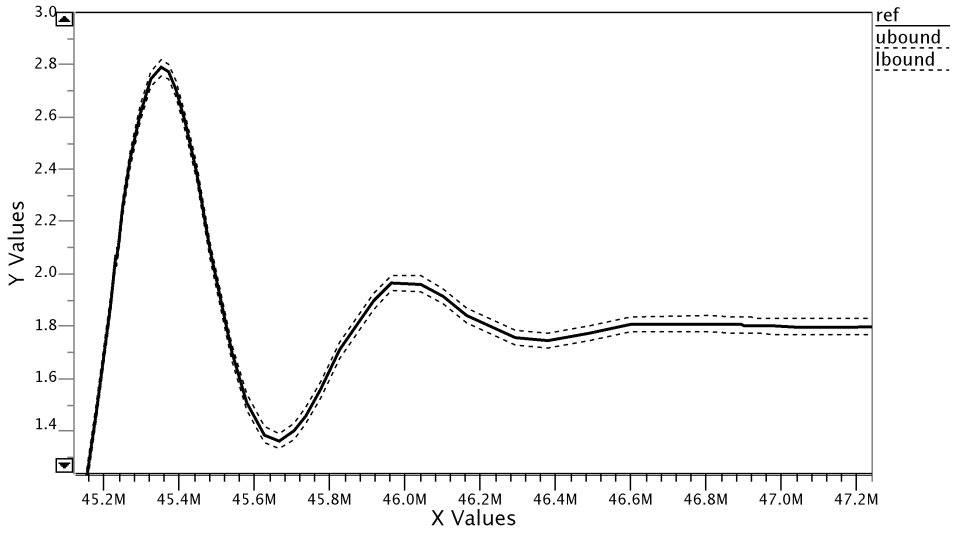
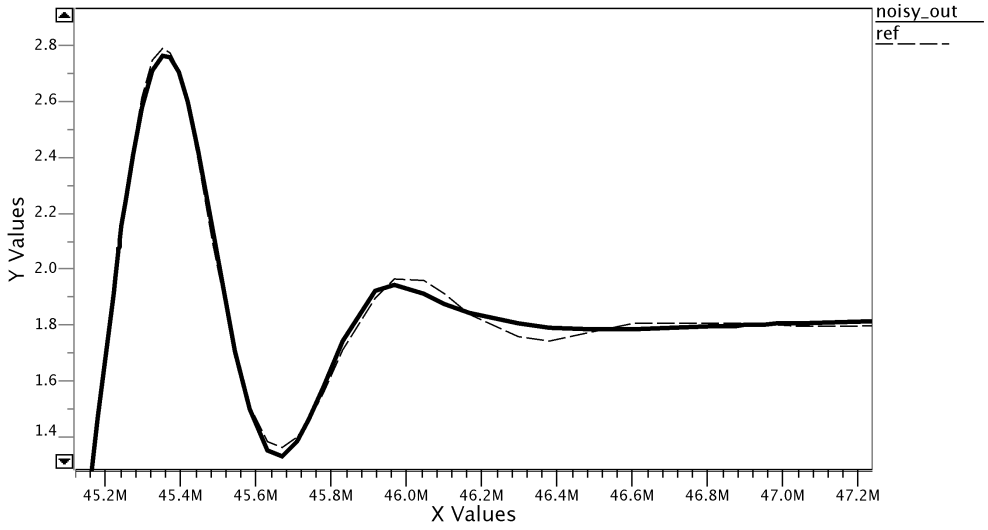


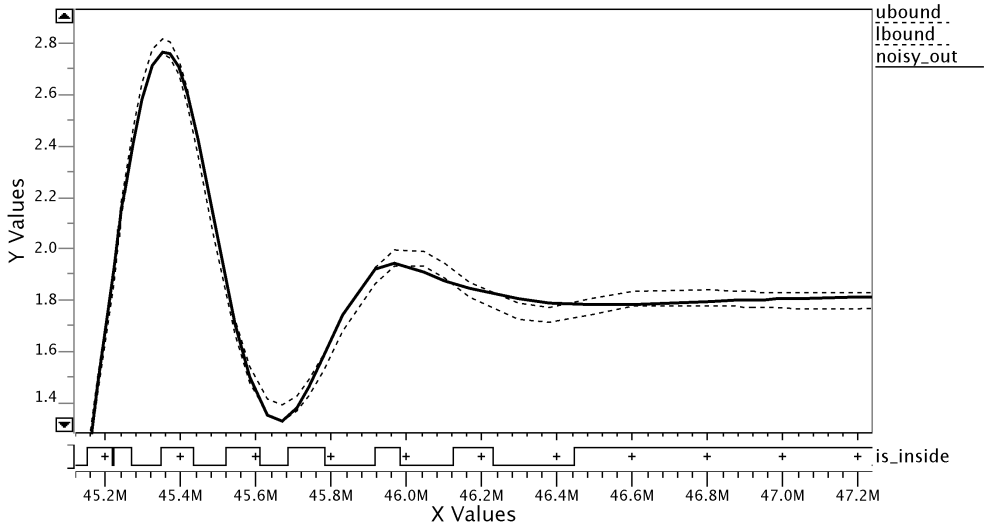
Figure 3.1. Haloes show tolerances for analog signals.

We show an analog comparison example in Figure 3.2. First, we plot a reference signal (denoted as *ref*) and a noisy test signal (denoted as *noisy_out*) in Figure 3.2a. In comparison of both signals, checking each data-point of both signals for equality does not work because analog irregularities make small differences between high-level and low-level simulations. Therefore, we have to define tolerances around reference signals. In Figure 3.2b, we compare same analog signals in Figure 3.2a while this time we create a halo for the reference signal to express a tolerance of ±30mV. Upper and lower boundaries of the halo are denoted as *ubound* and *lbound*, respectively. Comparison operator checks whether test signal is inside the halo and returns a Boolean signal denoted as *is_inside*.

This way we relax analog comparison with applying tolerated regions around analog signals via haloes, which is closer to the understanding of an analog designer. It is an important stage to automate analog verification and we believe any analog verification framework should treat analog tolerance values as first-class specifications. As a next step, we define tolerances in our analog verification environment. Therefore, we investigate possible ways to calculate a halo in next section.



(a) Exact comparison does not work for analog.



(b) Comparison with tolerances is natural.

Figure 3.2. Comparison with tolerances.

3.2. Halo Calculation

Halo calculation is the process of defining upper and a lower bounds to be used in analog comparisons. This way acceptable regions for analog signals are defined, and it allows us to comment on the equivalence of analog signals. Halo calculation process can be manual and automatic depending on the properties of analog signals in comparison, requirements of the application and designers' expertise. Dictating a single halo calculation method for all kinds of analog signals would fail or be inadequate for a complete AMS verification framework because this framework has to handle as many situations as possible. For example, the output of a DAC circuit is a stepped analog signal by design while op-amp is usually verified by observing sinusoidal input-output signals. Therefore, we study various ways to calculate haloes in the following sections.

3.2.1. Absolute Method

Absolute method to calculate haloes uses a user-defined constant to define tolerance values. Adding and subtracting user-defined constant value from the original signal value create upper and lower boundaries therefore it is a simple and straightforward way to calculate a halo. In this case, the shape of the boundaries is exactly the same as the signal.

We show an example halo calculation using absolute method in Figure 3.3. Reference signal (denoted as *ref*) is used as a base signal and upper boundary (denoted as *ubound*) and lower boundary (denoted as *lbound*) are created by applying tolerance values.

3.2.2. Relative Method

Relative method to calculate haloes uses a user-defined percentage to define tolerance values. Adding and subtracting user-defined percentage value from the original signal value creates upper and lower boundaries therefore it is another way to calculate a halo. In this case, the shape of the boundaries is scaled by the same as the

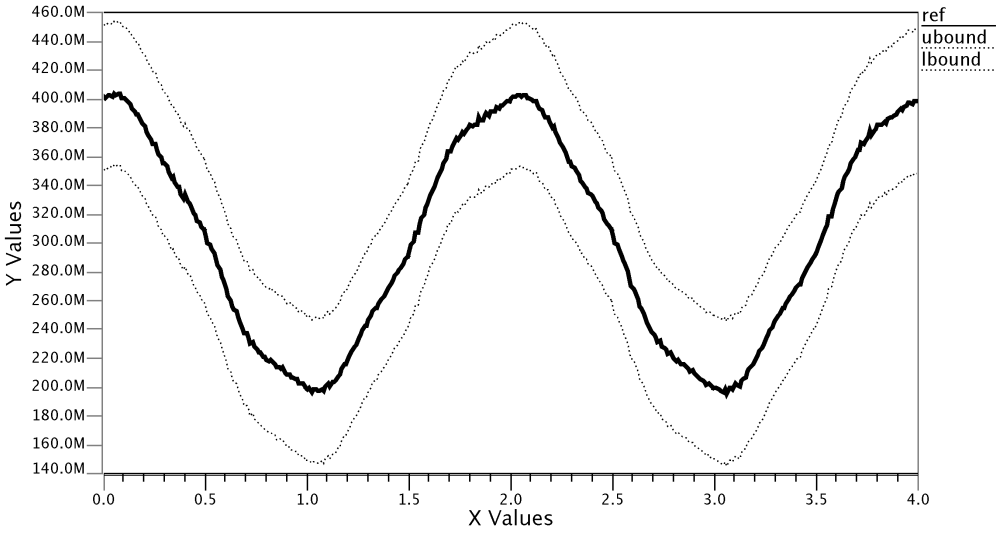


Figure 3.3. Halo calculated by an absolute tolerance value, which is 0.05V.

signal.

We show an example halo calculation using relative method in Figure 3.4. Reference signal (denoted as *ref*) is used as a base signal and upper boundary (denoted as *ubound*) and lower boundary (denoted as *lbound*) are created by applying tolerance values.

3.2.3. Smoothing Method

Smoothing is defined as creating an approximate representation that attempts to capture the general pattern in the signal, while leaving out noise or rapid fluctuations on the signal. Because analog signals are always associated with noise in real life, noise simulations are widely used to investigate inherent or external noise effects on analog circuits. Apart from noise simulations, AMS circuits usually involve spikes (rapid changes due to digital switching) on analog signals. This may result in a failure for automatic analog verification while an analog designer can simply ignore these spikes and focus on general pattern. By smoothing we aim the same effect for automatic

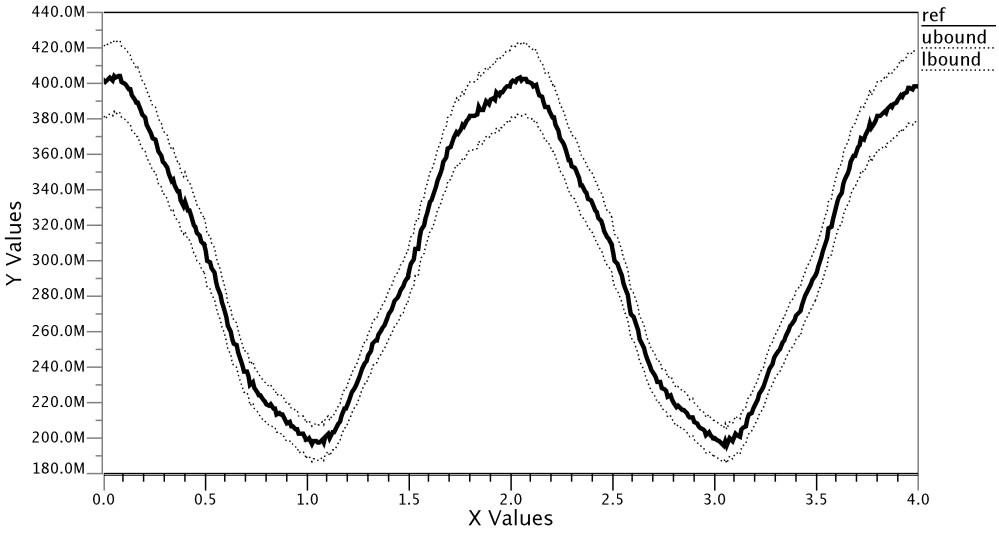


Figure 3.4. Halo calculated by a relative tolerance value, which is is 5%.

verification. Therefore, it is practical to use smoothed signal instead of original noisy signals and we can check properties on general patterns of analog signals.

Various algorithms are proposed to smooth one-dimensional signals in the literature. In the context of this thesis, we investigate low-pass filters, smoothing splines and Savitzky–Golay algorithm. These algorithms are provided by scientific libraires and we support them in our framework.

We show an example halo calculation using smoothing method in Figure 3.5. Reference signal (denoted as *ref*) is smoothed by using Savitzky–Golay with parameters (window_order=19, order=3) and smooth signal is used as a base signal and upper boundary (denoted as *ubound*) and lower boundary (denoted as *lbound*) are created by applying tolerance values.

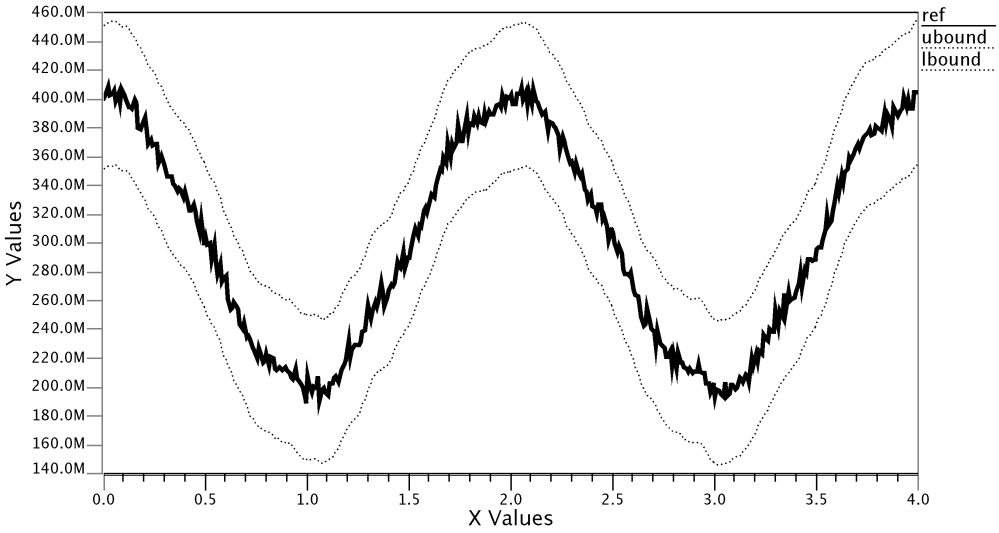


Figure 3.5. Halo calculated by smoothing reference signal. Tolerance value is 0.05V.

3.2.4. Monte Carlo Method

A number of analog signals obtained by Monte-Carlo simulations are used to calculate a halo. Simply maximum and minimum values of these signals for each sample point are considered as upper and lower boundaries, respectively.

3.3. Analog Comparison

3.3.1. Analog Signal Representation

Although analog signals are continuous in both time and value domains, we have to deal with their finite representations in computer systems. Analytically, analog signals are represented by mathematical functions mapping time domain to value domain in full extent. This means that an analog signal consists of infinite number of $(t, f(t))$ time-value pairs where t is ranging over an interval in time. However, due to finiteness of computer systems, analog signals have to be represented in computer systems by any subset of the sequence of such pairs, with a finite set of t values called

sampling points and this process is called *sampling*.

In practice, complex systems like analog circuits usually produce mathematical functions that are very hard to solve or unsolvable by analytical methods. In such case, numerical simulators can be used to generate desired outputs by including a trade-off between accuracy and computation time. Increasing accuracy means computing for more sample points and obviously increases computation time. If analog signals do not sample with sufficient accuracy, questions over analog signals can be easily resulted in wrong answers.

Sampling process creates an approximation of original function and some information has been lost as much as like any approximation. For example, values of analog signals at time instants remaining between any consequent sampling points t_1 and t_2 are virtually unknown. After sampling, analog signals becomes an object of numerical analysis rather than analytical analysis. Therefore, these unknown values between adjacent samples may be guessed again by some numerical techniques such as interpolation if needed.

3.3.2. Booleanization

We define analog (real-valued or continuous) signals as functions from the time-domain $\mathbb{T} = \mathbb{R}^+$ to a state-space $\mathbb{X} \subseteq \mathbb{R}^n$, where n is the number of predicates. The *Booleanization* of an analog signal corresponds to a transformation of the state-space $\mathbb{X}$ of the signal into Boolean values using a predicate such that $\mu : \mathbb{X} \to \mathbb{B}$. After using such predicates over analog signals, the analog signal $a : \mathbb{T} \to \mathbb{X}$ is transformed into a Boolean signal $w : \mathbb{T} \to \mathbb{B}$.

Rise and *Fall* events for Boolean signals correspond some qualitative changes in analog signals and a Boolean signal can be fully identified by its *Rise* and *Fall* events. Therefore, a Boolean signal is nothing but a sequence of *Rise* and *Fall* events. In our implementation, each Boolean event (*Rise* and *Fall*) in a Boolean signal is associated with a time-stamp. We prefer to use continuous-time notion to express time. This

means that *Rise* and *Fall* events can happen at any time and it is independent from any type of clocking mechanism.

Although we want to express Boolean signals in continuous-time, analog signals are represented by their sampled versions in practice. This means that a predicate that we use to booleanize an analog signal should return a continuous-time boolean signal from a sampled analog signal. For such a duty, we implement a predicate which determines if an analog signal is *Greater-than-zero* or not and returns a continuous-time Boolean signal. This is a base method for us, and all other comparison operators can be derived from this method. This method sequentially checks each sample value of the analog signal, and searches for a change in the output value. Whenever a change happens, it places the corresponding event (either *Rise* or *Fall*) at that time instant. For example, if the current signal sample value is positive, and the previous sample value is negative, this means that the output value of *Greater-than-zero* operator is changed at some time between the current and the previous samples. We use linear interpolation to calculate the exact zero-crossing time, denoted by $t_{ZeroCross}$, and we place a *Rise* event at the zero-cross instant. Similarly, we place a *Fall* event to the zero-cross instant if the previous sample value is positive, and the current sample

Table 3.1. Computation a boolean signal from sample values.

a_n	a_{n+1}	Timestamp	Event
–	+	$t_{ZeroCross}$	Rise
–	0	N/A	No Event
–	–	N/A	No Event
0	+	t_n	Rise
0	0	N/A	No Event
0	–	N/A	No Event
+	+	N/A	No Event
+	0	t_{n+1}	Fall
+	–	$t_{ZeroCross}$	Fall

value is negative. If there is no change at the output, no event is placed. All possible cases for successive analog samples a_n and a_{n+1}, where n denotes sample index. The corresponding time-stamp and event are displayed in Table 3.1.

In our AMS verification framework, booleanization predicates include regular comparison operators $\{<, >, \leq, and \geq\}$ except exact equality operator $\{=\}$. Instead we provide two convenient comparison operators, *inrange* and *compare* operators to check equivalence of signals. The operator *inrange* is meant to check if the signal is in the range of an interval defined by upper and lower bounds as well as the operator *compare* checks equivalence of two analog signals considering tolerance defined by haloes.

4. INTEGRATING MEASUREMENTS AND CIRCUIT ANALYSES

Measurement operations and circuit analyses are widely used ways to collect information about analog designs. Today's analog verification tasks are predominantly based on these measurements and analyses. Therefore, integrating measurements and circuit analysis would make assertion based AMS verification much more expressive and beneficial from analog point of view, and it is an important and required step for complete AMS verification flow.

In the context of this thesis, we use both measurements and circuit analyses as instantaneous observers for the design, that return a set of events. For example, crossing measurement for an analog signal observes the signal and returns a set of events that shows where the signal crosses a certain value. Similarly, DC analysis performed at specified time instants returns a set of events that shows DC voltages and currents of the circuit nodes. As we handle measurements and circuit analyses using event-based approach, we explain how we represent events in our AMS verification framework in Section 4.1. Then, we give details about measurements in Section 4.2 and about circuit analyses in Section 4.3.

4.1. Event Representation

An event describes a change in state. For our purposes, we think of an event as an instantaneous event, that means it occurs at a time instant. Then, we define an event e as an $(n+1)$-tuple, that is $e = (t, x_1, \ldots, x_n)$ where t is a time instant at which the event occurs and $x_1, \ldots, x_n$ denote n number of custom variable values related to the event. For example, rise-time measurement on an analog signal produces a set of events such as $E = [e_0, e_1, \ldots, e_m]$ where each event $e_m = (t_m, x_{rt,m})$ where t_m denotes the time of rise-time measurement and $x_{rt,m}$ denotes the value of rise-time measurement.

In Figure 4.1, we plot an example signal (denoted as *signal*) and and a set of

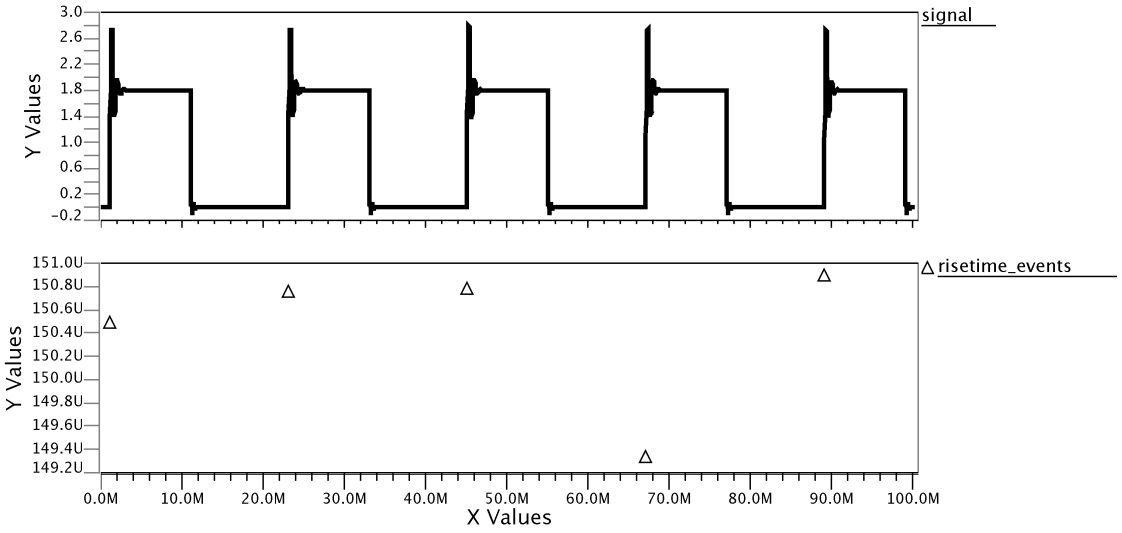


Figure 4.1. Events can represent measurement results for signals.

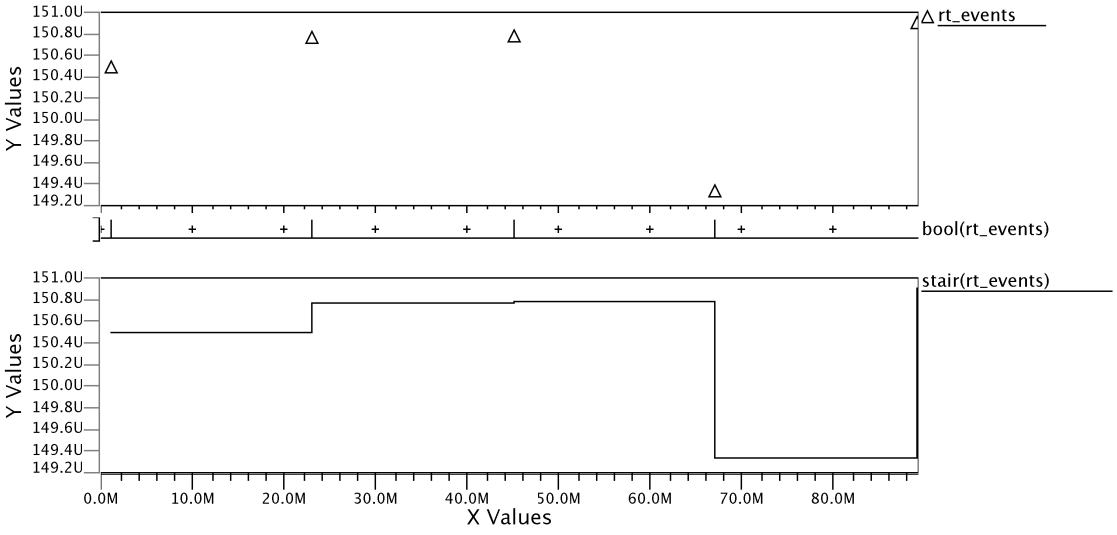


Figure 4.2. Events can be translated into Boolean and analog signals.

rise-time measurement events (denoted as *risetime_events*) computed from *signal*.

4.2. Measurements for AMS Assertions

Analog measurement functions are used to extract data from analog signals. In the context of assertions, extracted data can be checked to see whether they satisfy design specifications. Depending on analysis type, seeral measurement operations are listed below.

4.2.1. General Measurements

average

This measurement finds the average value of the specified signal.

crossing

This measurement finds the intersection points between a signal and reference value.

intersect

This measurement finds the intersection points between two signals.

max

This measurement finds the local maxima of the specified signal from the time of given events.

min

This measurement finds the local minima of the specified signal from the time of given events.

p2p

This measurement finds the peak-to-peak values of the specified signal.

slope

This measurement finds the slope values of the specified signal at the time of given events.

yval

This measurement finds the Y values at the time of given events.

4.2.2. Time Domain Measurements

delay

This measurement finds the delay between the edges on one or two waveforms relative to default (automatically calculated) or user-specified topline and baseline levels for both the measured waveform and the reference waveform.

dutycycle

This measurement finds the duty cycle of a periodic waveform relative to default (automatic calculated) or user-specified topline and baseline levels. The duty cycle of the periodic waveform is the ratio of the high portion of the waveform to the length of the period. The high portion of a cycle is the duration of the positive pulse measured at the middle level.

falltime

This measurement finds the falltime between specified upper and lower levels of a waveform. The falltime is calculated as the difference in time when the waveform falls from the upper level to the lower level.

frequency

This measurement finds the frequency of a periodic waveform relative to default or specified topline and baseline levels. The frequency is calculated as the reciprocal of the period.

overshoot

This measurement finds the overshoot value of a waveform. The overshoot value is calculated as the difference between the maximum point and the topline level of the waveform.

period

This measurement finds the period of a periodic waveform relative to default or specified topline and baseline levels. The period is calculated as the difference in time between two consecutive edges of the waveform of the same polarity (i.e. rising edge to rising edge or falling edge to falling edge).

pulsewidth

This measurement finds the pulse width of a waveform relative to default or

specified topline and baseline levels. The pulse width, for a positive pulse, is the difference in time between the middle level of a rising edge and the middle level of the next falling edge on the waveform. For a negative pulse, the pulse is the time difference between the middle level of a falling edge and the middle level of the next rising edge.

risetime

This measurement finds the risetime between selected upper/lower levels of a waveform. The risetime is the difference in time when the waveform rises from the lower level to the upper level.

slewrate

This measurement finds the slew rate of the waveform. The slew rate is the difference between the upper and lower levels of the waveform divided by the risetime of the rising edge (or the falltime of the falling edge).

undershoot

This measurement finds the undershoot value of a waveform. The undershoot value is calculated as the difference between the minimum point and the baseline level of the waveform.

4.2.3. Frequency Domain Measurements

bandwidth

This measurement finds the bandwidth, the lower band edge, upper band edge, center frequency and quality factor, and the level at which the measurement is made for a bandpass-shaped waveform.

gainmargin

This measurement finds the gain margin in decibels (dB) and the associated crossover frequencies of a complex waveform. The gain margin is defined as the difference between the gain of the measured waveform and 0 dB (unity gain) at the frequency where the phase shift is -180 degrees.

phasemargin

This measurement finds the phase margin of a complex waveform in degrees or

radians. The phase margin is defined as the difference in phase between the measured waveform and -180 degrees at the point corresponding to the frequency that gives us a gain of 0 dB.

4.3. Circuit Analyses

Circuit analyses are well-defined ways of collecting information (metrics) about analog designs. Because circuit analyses provide valuable metrics for verification, we see a great benefit to integrate them into assertion-based mixed-signal verification. Among circuit analyses, DC operating point analysis determines the quiescent state of circuits. AC analysis extracts small-signal linear response of the circuit for a single frequency input around the DC operating point. Transient analysis computes time-domain response of the circuit by iteratively solving the algebraic differential system of equations. Fourier analysis, by using Fast Fourier Transform (FFT), returns power spectrum of time-domain signal so that we can see how much power resides in frequency components of that signal. We can analyze noise and linearity characteristics of analog circuits from power spectra. In a mixed-signal design environment, digital verification techniques like assertions are still available on top of circuit analyses. In this paper, we focus on DC, AC and Fourier analyses and their integration into AMS assertion languages. We use transient analysis, which is essentially time-domain simulation of analog circuits, as a bridge between DC, AC and Fourier analyses in the verification of programmable analog circuits. It means that we start a transient analysis and we pause transient analysis whenever a digital event changing the state of analog circuit occurs. Then, we check DC, AC or Fourier characteristics of the circuit for that time instant. This way, we can monitor and verify time-varying characteristics of programmable analog circuits. We now describe each of these circuit analyses in following sections.

4.3.1. DC Operating Point Analysis and DC Assertions

DC operating point analysis determines the quiescent state or stable initial condition of the circuit. The quiescent state is computed by the simulator with capacitances opened and inductances short-circuited. It provides operating point

information before a small-signal (AC) analysis or a transient analysis so it is the starting point in the analog design flow.

In regular flow of analog design, designers make assumptions and choices for DC values inside their analog blocks. Assertions capture these assumptions and design choices and they report if there is undesired situation. DC characteristics of analog circuits include DC voltage levels of circuit nodes, DC current values for circuit branches as well as operating modes and conditions for circuit devices. Related analog circuit specifications that can be extracted via DC analysis include offset voltage values, offset voltage drifts, bias current values, current drifts and operation modes of transistor devices. For example, in Snippet S1, *node* denotes a circuit node in actual circuit and $@DC()$ construct extracts DC value of *node* at the start of simulation denoted by event e_{start}.

$$(node)@DC(e_{start}) \tag{S1}$$

For time-invariant analog circuit blocks, DC analysis is performed at the start of simulation and it is valid for all simulation times. Therefore, initial DC analysis is sufficient to verify DC characteristics for time-invariant systems. However, in case of circuits having time-varying DC characteristics such as adaptive or digitally controlled analog circuits, it is beneficial to integrate DC analysis checks into time-domain simulation (transient analysis). This way, the interaction between digital and analog domains can be investigated in a single simulation environment. To ensure desired DC characteristics during such a mixed-signal simulation, we integrate DC analysis checks into AMS assertions. For example, we want to check DC voltage level for the output node of an amplifier with digitally-adjustable transconductance. Because transconductance adjustment via digital inputs can shift DC voltage levels in the analog circuit, output DC voltage level $Vout$ should be monitored and checked during all time-domain simulation. In Assertion 4.2, we check whether DC values of node $Vout$ is always between 0.85V and 0.95V where consequential DC analyses performed at time

$e_{digital}$ events occur.

Assertion

$$Assert: \qquad \mathcal{G}(\ 0.85V < (\ Vout\ )@DC(e_{digital}) < 0.95V\ ) \tag{4.2}$$

end

4.3.2. AC Analysis and AC Assertions

Traditionally, designers analyze analog circuits by exciting them with a single-frequency input signal, and they measure the output signal to see how the magnitude and phase of the input signal is changed by the circuit. In general, only linear analog circuits can be analyzed using this technique; therefore, a nonlinear circuit should be linearized by assuming input signals are small enough. This technique, called as small-signal analysis or AC analysis, is a common and useful procedure when analyzing frequency response of analog circuits.

Related analog circuit specifications that can be extracted via AC analysis include DC gain, bandwidth, phase margin, gain margin and roll-off slopes.

In Figure 4.3, we show an example AC analysis plot where we can see measured AC metrics for the output node of a low-pass filter circuit. We can write assertions to check whether measured metrics satisfy the specifications in assertion-based verification. For example, if desired value for DC gain is greater than 60dB for an amplifier design, this specification is captured in following statement:

$$dcgain(\ Vout\ )@AC(e_{start}) > 60.0\text{dB} \tag{S2}$$

In Snippet S2, *Vout* denotes amplifier output node in actual circuit, and we first calculate dcgain of amplifier from AC analysis at the start of simulation denoted by event e_{start}, then dcgain value is compared with specification.

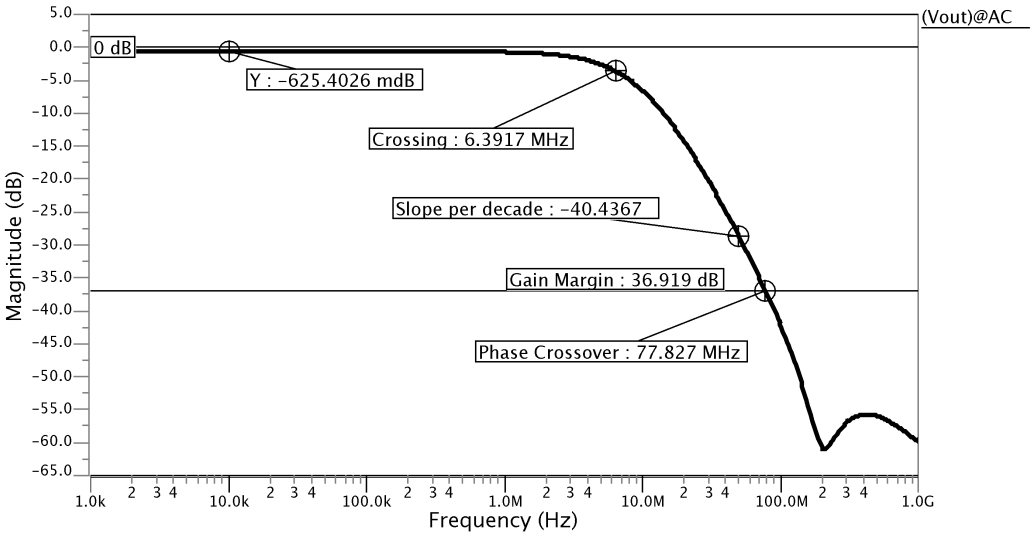


Figure 4.3. An example AC analysis plot and measured AC specifications.

For linear time-invariant (LTI) analog blocks, AC analysis is performed at the start of simulation after initial DC analysis and it is valid for all simulation times. Therefore, initial AC analysis is sufficient to verify AC characteristics for LTI systems. However, in case of circuits having time-varying AC characteristics such as adaptive or digitally controlled analog circuits, it is beneficial to integrate AC analysis checks into time-domain simulation (transient analysis) as we did for DC assertions. For example, if we want to check DC gain value for a low-pass filter with digitally-adjustable cut-off frequency, then we write the following assertion to capture DC gain property. where e_k denotes discrete events such as changes in digital control inputs.

Assertion

$$Assert: \qquad \mathcal{G}\left(-0.5 < dcgain(\ Vout\ )@AC(e_k) < 0.0\right) \tag{4.3}$$

end

In Figure 4.4, we illustrate how AC analysis is performed in the middle of a

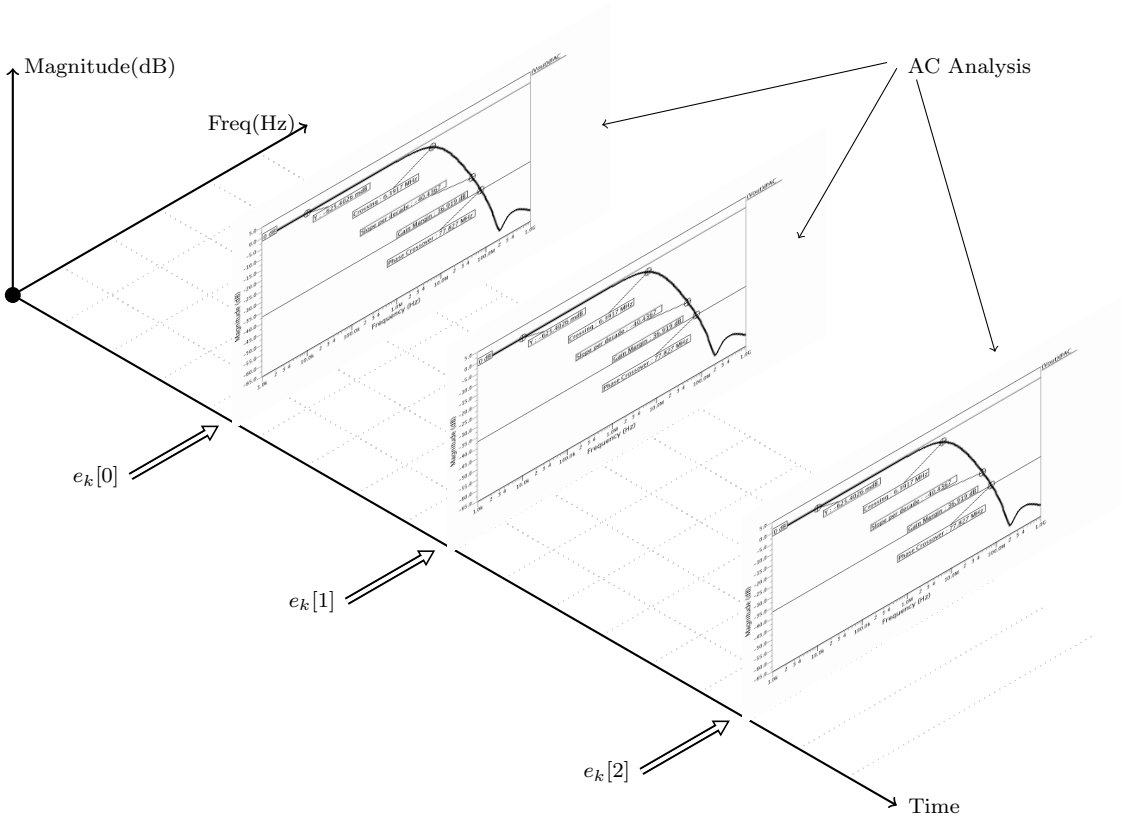


Figure 4.4. Monitoring time-varying AC characteristics in the middle of time-domain circuit simulation.

time-domain simulation. During time-domain simulation, AC analysis is performed for time instants when events e_1, e_2, e_3 occur. Then, our assertion monitors DC gain at the output node from AC analysis results and checks if it always satisfies the desired condition when these events occur.

An important limitation is as follows. Although current simulators can perform AC analysis in the middle of transient analysis, they use current node voltages for AC analysis instead of performing true DC analysis. This limitation makes each AC analysis which are performed at exact event time wrong. Therefore, we solve this problem practically by delaying corresponding event times (until DC voltages are stabilized) for AC analyses. With simulators allowing event-driven DC and AC analysis, analysis and verification of time-varying analog circuits are easier and more comfortable.

4.3.3. Fourier Analysis for Noise and Linearity

Noise and linearity characteristics of analog circuits determine the range of useful signals that the circuit processes as intended. The smallest value of signals that can occur is limited by the noise floor of the circuit whereas the the largest value of signals is limited by nonlinearity of the circuit. Therefore, we use Fast Fourier Transform (FFT) based analyses to verify dynamic range of analog circuits in practice.

Dynamic range metrics include total harmonic distortion (THD), signal-to-noise and distortion ratio (SNDR) and spurious free dynamic range (SFDR). We can extract these metrics by performing a FFT to get the power spectrum of time-domain simulation results if the circuit is excited with a single-frequency input. On the other hand, we should consider all factors to compute a healthy FFT such as number of points, windowing choice and sampling frequency during these analyses.

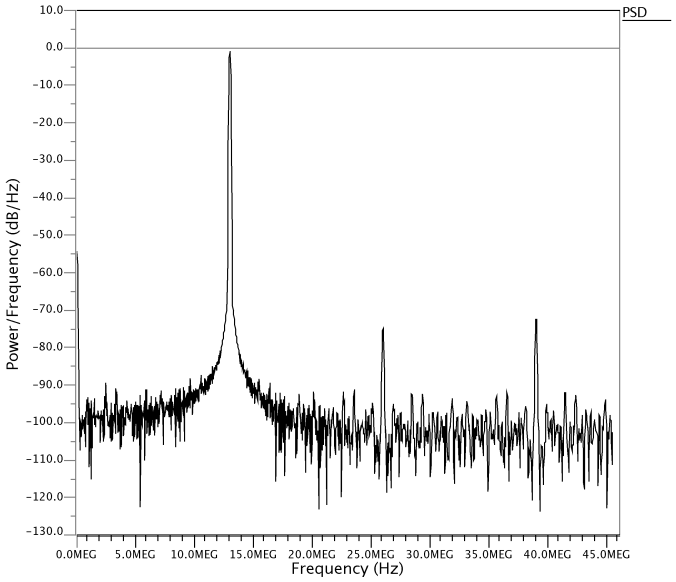


Figure 4.5. Power Spectrum Example.

In Figure 4.5, we illustrate an example power spectrum plot, which shows the power of each frequency component along frequency axis. The biggest peak implies the

fundamental frequency (13MHz in this case), where most of the power is concentrated, and we can see noise and distortion components in other frequencies.

The total harmonic distortion (THD) of a signal is the ratio of the total power of all second and higher harmonic components to the power of the fundamental harmonic (first harmonic) for that signal. Because a nonlinear system produces second, third and higher-order distortion components at the harmonics of the input (fundamental) frequency, when excited with a sinusoidal source, it is used as measurement for the nonlinearity of a system. THD metric is calculated by the formula below, and is expressed in gain (dB) or percentage.

$$\text{THD} = 10 \log \frac{H_{D2}^2 + H_{D3}^2 + \cdots + H_{Dn}^2}{H_{D1}^2}$$

where H_{D1}^2, H_{D2}^2, H_{D2}^2 and H_{Dn}^2 represent power value of first-, second-, third- and n-th order harmonics, respectively.

In Figure 4.6, we illustrate an example THD analysis of an analog circuit, which is excited by an input frequency of 1.3 MHz. In the normalized power spectrum of output signal, we can see the biggest peak at 1.3 MHz, which is the fundamental frequency, and a few smaller peaks at the integer-multiples of fundamental frequency, which are called harmonics. Then, THD operator calculates the ratio of the power of fundamental frequency and the power of all other harmonics. We write an assertion to monitor time-varying THD characteristics of any weakly nonlinear analog circuit as follows:

Assertion

$$Assert: \qquad \mathcal{G}\left(THD(\ Vout\ )@FFT(e_k) < 2.5\right) \tag{4.4}$$

end

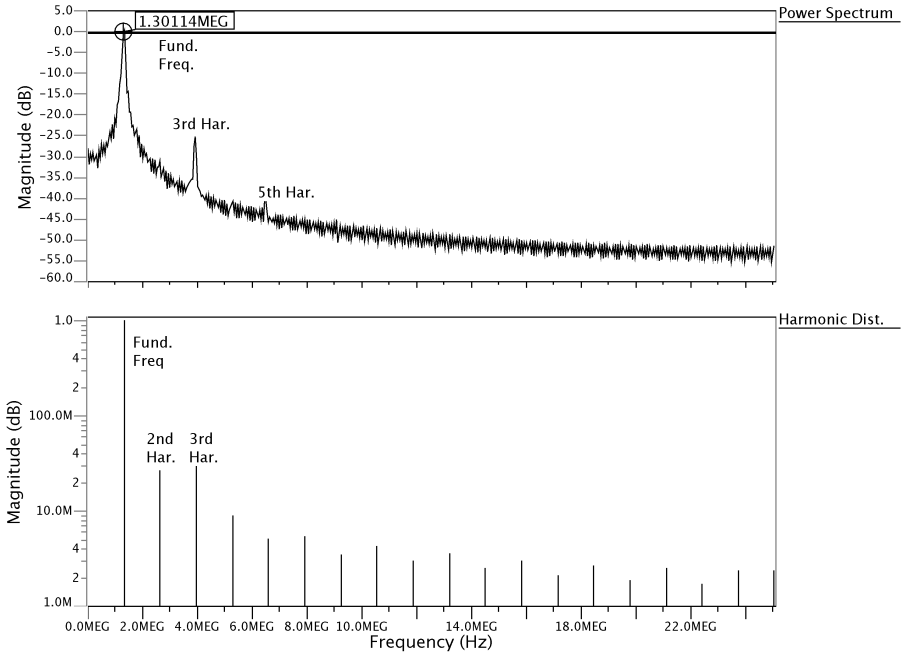


Figure 4.6. THD Example.

This assertion checks if calculated THD value always satisfies the desired specification at the time instants when event e_k occurs.

Another widely-used metric for noise and linearity specification is Signal-to-Noise-and-Distortion Ratio (SNDR). It is the ratio of signal power to power sum of all other spectral components, and expressed in dB. SNDR is a good indicator about system's performance because it takes both distortion and noise components into account. SNDR is calculated by using the formula below:

$$\text{SNDR} = 10 \log \frac{P_{signal}}{P_{noise} + P_{distortion}}$$

where P_{signal}, P_{noise} and $P_{distortion}$ denote the value of power of signal, noise and distortion components. We write an assertion to monitor time-varying SNDR

characteristics of any weakly nonlinear analog circuit as follows:

Assertion

$$Assert: \qquad \mathcal{G}\left(SNDR(\ Vout\ )@FFT(e_k) > 30dB\right) \tag{4.5}$$

end

This assertion checks if calculated SNDR value always satisfies the desired specification at the time instants when event e_k occurs.

Spurious-free dynamic range (SFDR) is the ratio of the input signal to the peak spurious component. Spurs can occur at the harmonics of fundamental frequency because of nonlinearity or at other frequency values because of mismatches in the circuit. In Figure 4.7, we illustrate an example SFDR analysis of an analog circuit. In the normalized power spectrum, we see that the ratio of the input signal power and the power of biggest peak component (third harmonic) is 23 dB.

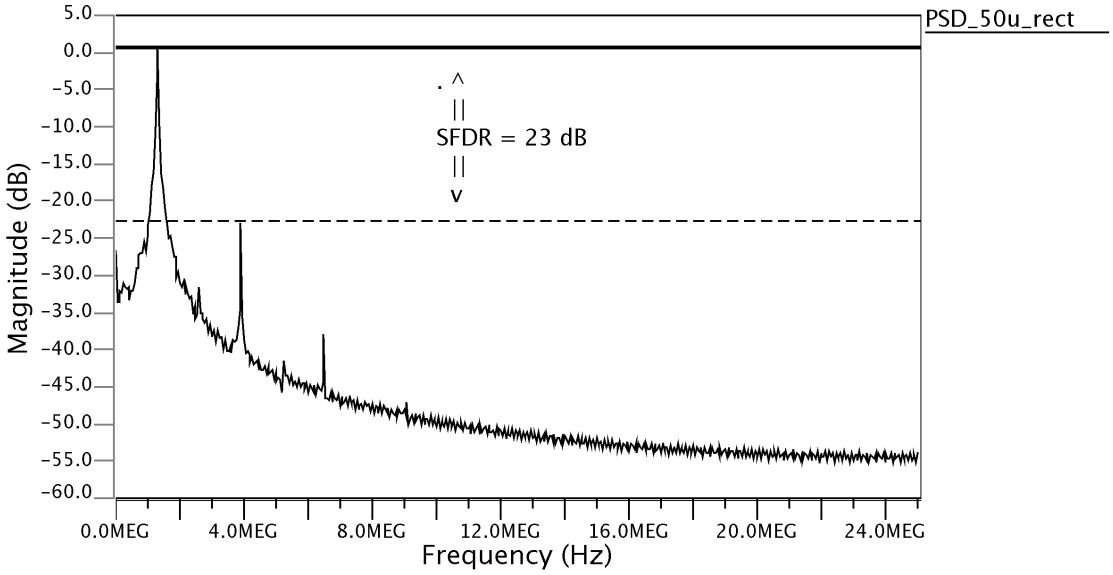


Figure 4.7. SFDR Example.

We write an assertion to monitor time-varying SFDR characteristics of any weakly nonlinear analog circuit as follows:

Assertion

$$Assert: \qquad \mathcal{G}\left(SFDR(\ Vout\ )@FFT(e_k) < 2.5\right) \tag{4.6}$$

end

This assertion checks if calculated SFDR value always satisfies the desired specification at the time instants when the event e_k occurred.

5. AMS-VERIFY FRAMEWORK

In this chapter, we demonstrate AMS-Verify framework that provides a unified environment for assertion-based AMS verification flow. The framework is written in Python language, and consists of three main elements: (i) Assertion checker, (ii) Waveform calculator and (iii) Scientific Python libraries.

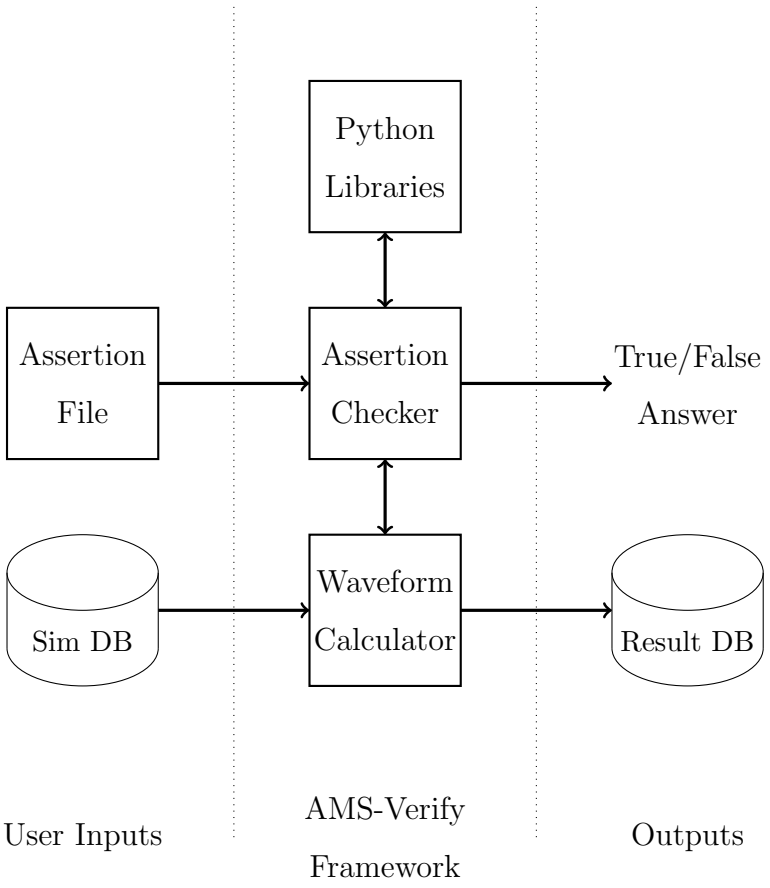


Figure 5.1. Overview of AMS-Verify Framework.

Assertion checker needs an input file specifying assertions to be checked and we explain details of input file structure in Section 5.1. After assertion checking process is explained in Section 5.2, a True/False answer is produced. Waveform calculator processes simulation data according to commands coming from the assertion checker, and produces a database including results and evaluation steps. This is handy to understand why and where an assertion fails. We developed waveform calculator as

a separate unit to allow to work with different waveform calculators, and initially we only support EzWave [41].

Finally, Python offers several scientific libraries as explained in [42], which can be very useful for high-level analog and mixed-signal verification. In critical parts of our framework, we employ scientific routines such as signal smoothing and Fast Fourier Transform (FFT) algorithms. Existence of these scientific libraries is the determinant factor for the language selection for our framework besides flexible and clean syntax of Python. Therefore, users are free to use these scientific libraries and standard Python constructs in assertion specification.

```
from amsverify.assertionsuite import AssertionSuite

class ExampleAssertionSuite(AssertionSuite):

    def setUp(self):
        self.dataset_open("example.wdb")
        self.wave = self.wf("<example>wave")

    def verify_in20mVrange(self):
        bdiff = self.inrange(self.wave, ubound=20e-3, lbound=-20e-3)
        self.assert_always(bdiff)

    def tearDown(self):
        self.dataset_save()

if __name__ == '__main__':
    ExampleAssertionSuite().run("verify_in20mVrange")
```

Figure 5.2. Example input file.

5.1. Input File Structure

Input files specifying assertions are regular Python source files and the input file structure is inspired from `unittest` package, that is a standard and popular unit testing framework of Python language. We show an input file example in Figure 5.2.

Similar to software unit testing frameworks, we developed a base assertion suite `amsverify.AssertionSuite` to verify properties of AMS designs. Individual assertions are defined with methods whose names start with the letters `verify`. Therefore, it is possible to define different assertions in a single assertion suite. The decisive point of each assertion is a call to `assert_always()` to check for condition that is desired to be always true. When a `setUp()` method is defined, the assertion checker will run that method prior to each assertion check. Likewise, if a `tearDown()` method is defined, the assertion checker will invoke that method after each assertion check. In case an assertion suite consists of more than one assertion, `setUp()` and `tearDown()` methods are executed for all assertions.

Assertion

$$Assert: \qquad \mathcal{G}(-20mV < wave < 20mV) \tag{5.7}$$

end

In Figure 5.2, we create an assertion suite (named `ExampleAssertionSuite`) with one assertion method called `verify_in20mVrange()`. This assertion method implements Assertion 5.7, that checks whether the signal *wave* is always in $20mV$ and $-20mV$ range or not. Next section will give more details about the evaluation of this assertion.

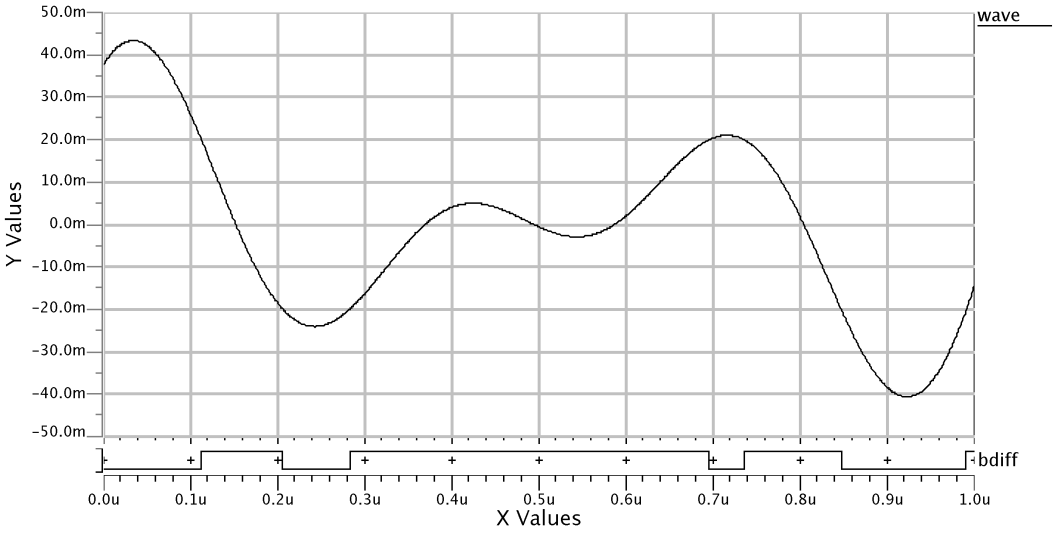


Figure 5.3. Signals for assertion evaluation of input file in Figure 5.2.

5.2. Assertion Evaluation

After we specify assertions in the format explained in Section 5.1, we evaluate assertions individually by calling `run()` method taking the name of assertion to be evaluated as its single argument (Default value is `"verify"`). Evaluation process starts by invoking `setUp()` method of assertion suite and `setUp()` method is meant to prepare environment before actual assertion, which can be common for all assertions such as loading a signal or calculating some constants. If `setUp()` method is not defined, it would be an empty method. Similarly, `tearDown()` method is called after evaluation process, which is meant to perform cleaning tasks such as saving such signals if necessary. If `tearDown()` method is not defined, it would be an empty method.

Between `setUp()` and `tearDown()` methods, actual assertion is evaluated. As it can be seen in Figure 5.2, we first open a dataset including simulation results and load the signal `wave` corresponding to *wave* in Assertion 5.7. `inrange()` method processes the signal `wave` according to the specification and returns a Boolean signal called `bdiff`. We plot these signals in Figure 5.3. `bdiff` indicates that `wave` is not always in

specified range therefore `assert_always()` method would return false. As the output of this assertion, True/False answer in Figure 5.4 is returned.

```
>>> verify_in20mVrange ... False (Elapsed time: 0.071368932724s)
```

Figure 5.4. Output of assertion evaluation of input file in Figure 5.2.

6. CASE STUDIES

In this chapter, we investigate three case studies intended to demonstrate the usefulness of assertion-based approach for checking the correctness of analog and mixed-signal designs with a focus on our contribution to this field. The first case study is described in Section 6.1 and it involves settling time property for two example signals. The second case study is explained in Section 6.2 and it aims to partially verify correct operation inside a real two-stage pipeline ADC design, which is presented in [43]. The third case study addresses a specific family of AMS circuits, namely programmable analog circuits, and a programmable Gm-C filter design is studied. All simulations were carried out with Mentor Graphics ADMS and Eldo simulators on a 2.67 GHz Intel-Xeon server with 8GB RAM.

6.1. Settling Time Property

We check settling time property in this case study. This property ensures that an analog signal enters a specified tolerance region and remains there after a rising/falling event occurs for the analog signal. In modern AMS design, interconnects between high-speed blocks suffer signal integrity related problems. A signal with good integrity means that the signal has clean and fast transitions. Increasing analog effects for advanced process nodes makes interconnects more likely to have ringing effects causing settling time increase.

Settling time property states that the signal should settle into a tolerance region (determined by tolerance parameters) around settling voltage value within specified time (called settling time) after rise/fall transitions. A signal is considered as settled if it remains inside tolerance region for a given time called settling duration. For this case, tolerance value is 0.01V, settling time is 7.5% of signal frequency, thus 1.115ns and settling voltage is 1.8V and 0.0V for rise events and fall events, respectively. Settling duration is given as 0.5ns. To check this property on simulation data in our framework, we first have to translate natural language specification into a formal

language. Therefore, in Assertion 6.8, we capture settling time property using our assertion language.

Assertion

$$Assert: \qquad \mathcal{G}(r \wedge f) \tag{6.8}$$

where,

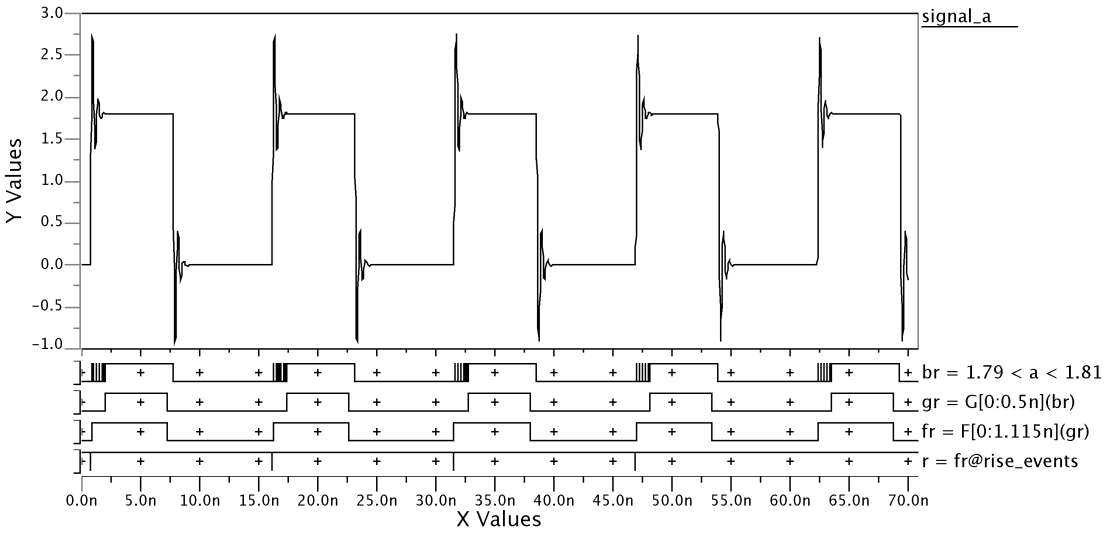
$$r = \left\{\mathcal{F}_{\ [0:1.115n]}\, \mathcal{G}_{\ [0:0.5n]}(1.79 < a < 1.81)\right\} @(crossing(a, 0.9, "rising"))$$
$$f = \left\{\mathcal{F}_{\ [0:1.115n]}\, \mathcal{G}_{\ [0:0.5n]}(0.01 < a < -0.01)\right\} @(crossing(a, 0.9, "falling"))$$

end

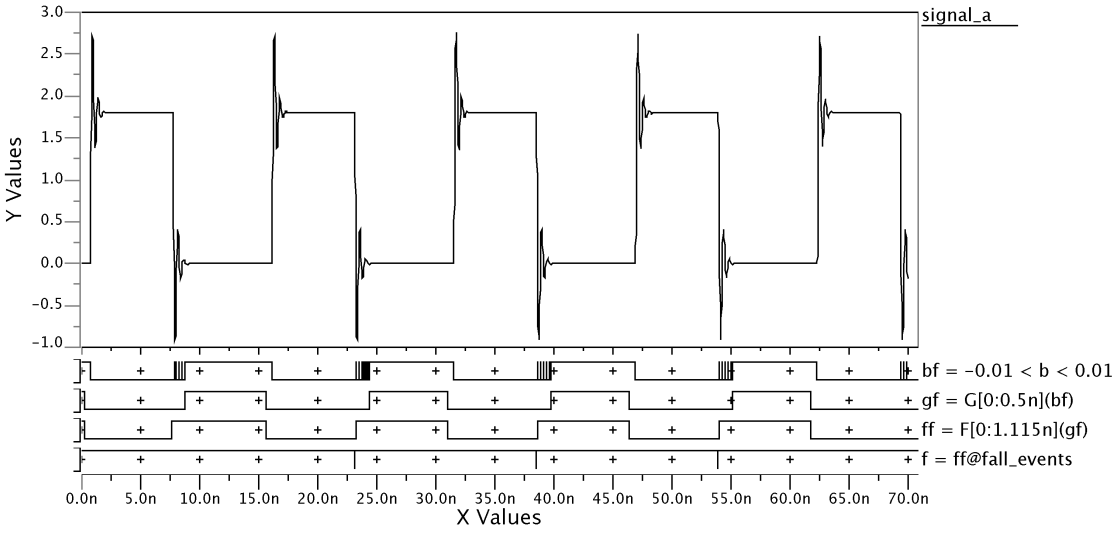
In Assertion 6.8, we divide settling time property into two subformulas, r and f denoting results of rising and falling checks. The assertion returns true if $r \wedge f$ is always true. Each subformula has same structure and only differences are desired settling regions and event definitions. Rise transitions should settle to 1.8V and fall transitions should settle to 0.0V with allowed tolerances. We plot evaluation results of subformula r and f in Figure 6.1.

We can see that comparison operator checks if the signal is inside specified region and returns Boolean signals br and bf for rising and falling transitions. Then *Always* operator $\mathcal{G}$ checks whether the signal remains inside the region at least for 1.5ns. Resulting signals are denoted as gr and gf. *Eventually* operator $\mathcal{F}$ returns fr and ff. It checks after the signal is settled at most after 1.115ns from corresponding event, where events are marked with event operator @.

In Figure 6.1, we can see that r and f is not true for all time instants, that means this assertion is failed and this design does not satisfy the settling time property. As a possible user scenario, next step would be to try to reduce ringing effect to satisfy specification, we have improved our design and simulated again, and we check the new

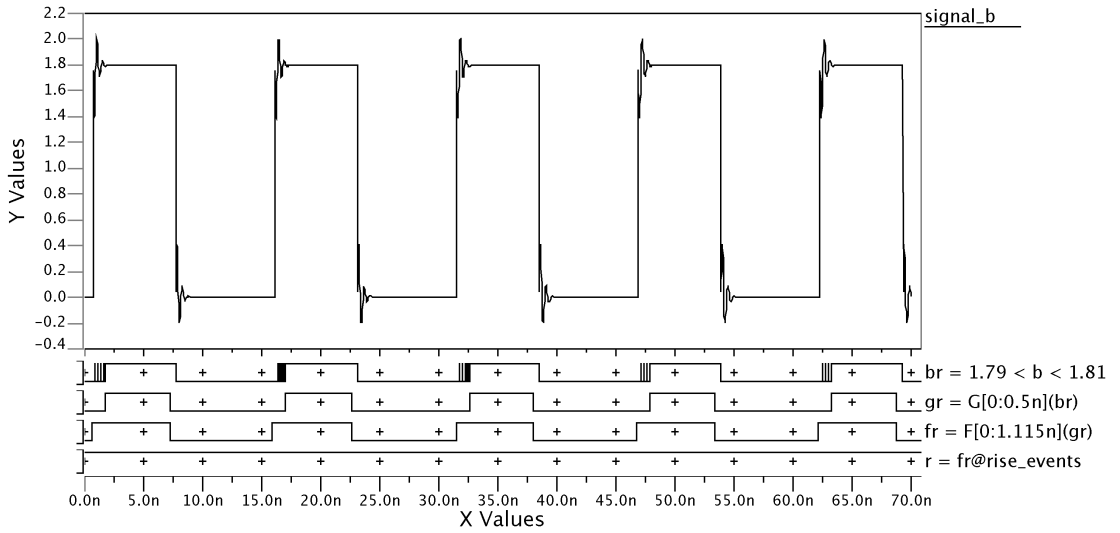


(a) Evaluation results of subformula r for the signal a.

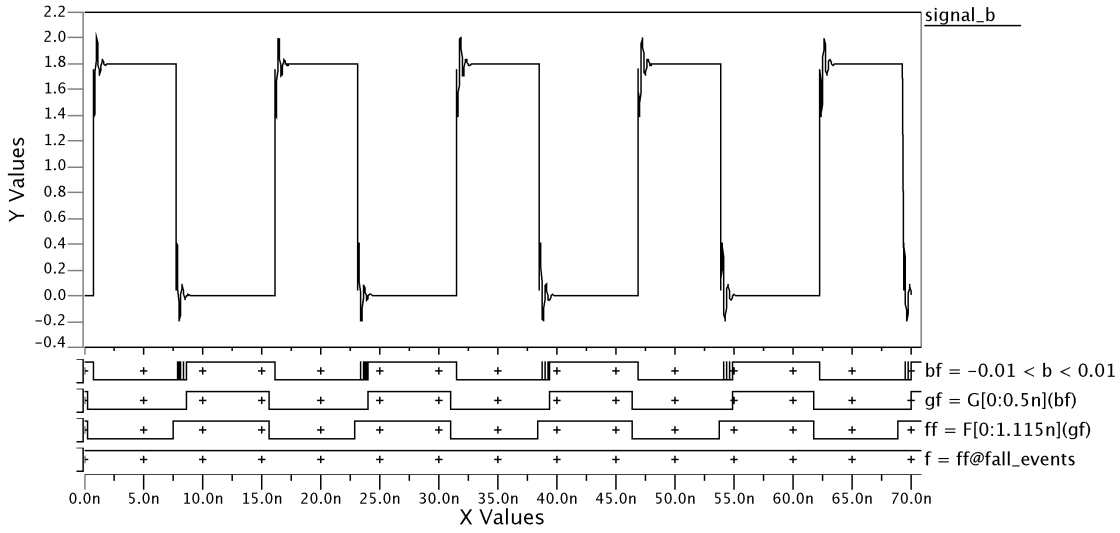


(b) Evaluation results of subformula f for the signal a.

Figure 6.1. Evaluation results of subformula r and f for the signal a.



(a) Evaluation results of subformula r for the signal b.



(b) Evaluation results of subformula f for the signal b.

Figure 6.2. Evaluation results of subformula r and f for the signal b.

signal b with the same assertion. In Figure 6.2, we plot the signal b and evaluation steps of same operations as in Figure 6.1. However, this time r and f is evaluated as always true so overall assertion is satisfied.

Settling time property reveals that we often write assertions by using similar constructs with a few tweaks such as parameter changes. Therefore, a flexible framework to allow defining parameterized assertions as well as simple scripting constructs such as local variables or loops. We see a practical need to include such scripting constructs in our AMS verification framework.

6.2. SH and DAC

We check the relation between the output of sample-n-hold (SH) block and the output of digital-to-analog converter (DAC) block in this case study. Studied SH and DAC blocks are part of a 10-bit two-stage pipelined flash analog-to-digital converter (ADC) design whose architecture is explained in [43].

In studied ADC design, the input voltage v_{in} is first sampled and held steady by a sample-and-hold circuit, obtaining the signal *sh*. For the first stage of the pipeline, first flash ADC converts signal *sh* to a 5-bit digital value, which is the first 5-bits of overall ADC output, out_{ADC}. The 5-bit value is then fed to a digital-to-analog converter obtaining signal *dac*, and this signal is subtracted from v_{in}. This residue is then fed to the second stage of the pipeline to obtain the last 5-bits of out_{ADC}. Whole ADC design is designed for differential input and output; therefore $V_{ss} = -0.9V$ and $V_{dd} = 0.9$. Input range of the ADC design is between $-0.45V$ and $0.45V$. In Figure 6.3, we show analog signals, *sh* and *dac*, and rising edge-triggered clock signal, *clk*, which are used in the ADC design experiments.

According to the concept of two-stage pipelined ADC design, input signal is sampled and held via SH circuit, and output of SH circuit is digitized by ADC and reconstructed by DAC again. Therefore, output of SH circuit and output of DAC circuit should be equal for the same sample. However, DAC output should be truncated

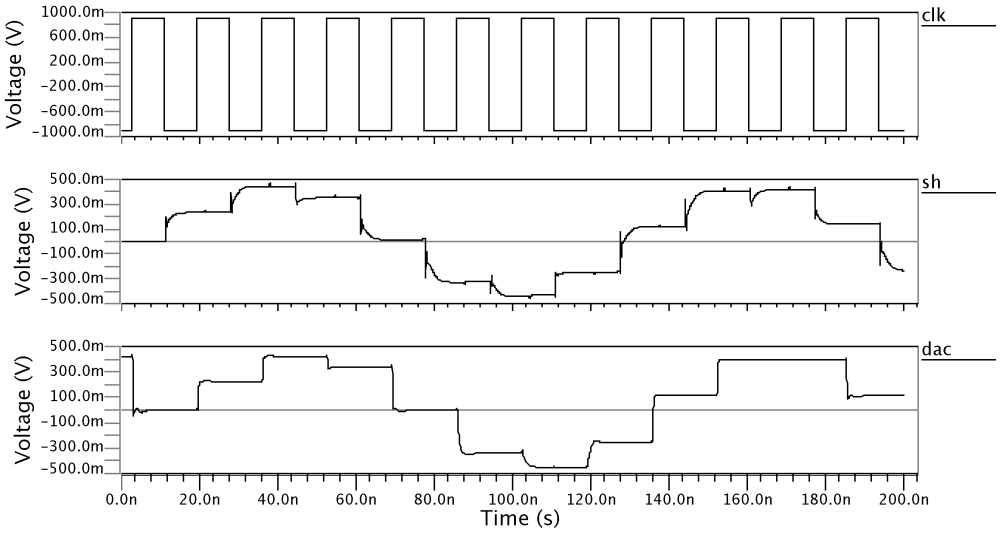


Figure 6.3. Output signals of SH and DAC circuit as well as triggering clock signal.

to correct quantization levels rather than exact SH output. In this case study, we want to check this relation between SH and DAC outputs, and we capture this relation in Assertion 6.9.

Assertion

$$Assert: \quad \mathcal{G}\left\{\left\{\mathcal{F}_{\ [0:12n]}\,\mathcal{G}_{\ [0:5n]}\, error\right\}@(crossing(clk, 0.0, "rising"))\right\} \tag{6.9}$$

where,

$$error = compare(dac,\ dacref,\ "absolute", tol = 1m)$$
$$dacref = datatowf(refdata)$$

end

In Assertion 6.9, we first numerically calculate reference data (*refdata*) for DAC output using SH values at the time of rising clock edge. Comparison method *compare*

compares actual simulated DAC output *dac* and *refdata* according to given tolerance calculation method and parameters. Bounded-time temporal operators $\mathcal{G}$ and $\mathcal{F}$ ensure that *dac* and *dacref* signals are equal when *dac* is settled after transition occurs. In Figure 6.4, we plot evaluation steps, and we can see asserted Boolean signal *rd* is not always true, which makes the assertion failed. Inspecting *rd* waveform reveals that it is false for two time intervals. First failure occurs at the start-up of simulation and analog circuit can have slow start-up conditions. Therefore, it can be ignored if start-up speed of the design is not critical. Assertion can be developed to include this relaxation so next-time we check the assertion (e.g. for other design) we do not get such a failure.

On the other hand, second failure points to a more crucial error, which is that DAC circuit is not able give an appropriate output for very bottom quantization level (in this case, DAC gives $-0.42V$ instead of $-0.45V$ which is outside of $1mV$ tolerance.). Solving this problem may not be trivial for the designer, however formalizing desired property and automatizing of evaluation of simulation results via assertions would definitely speed-up the design and allow designers to try different solutions in a shorter time.

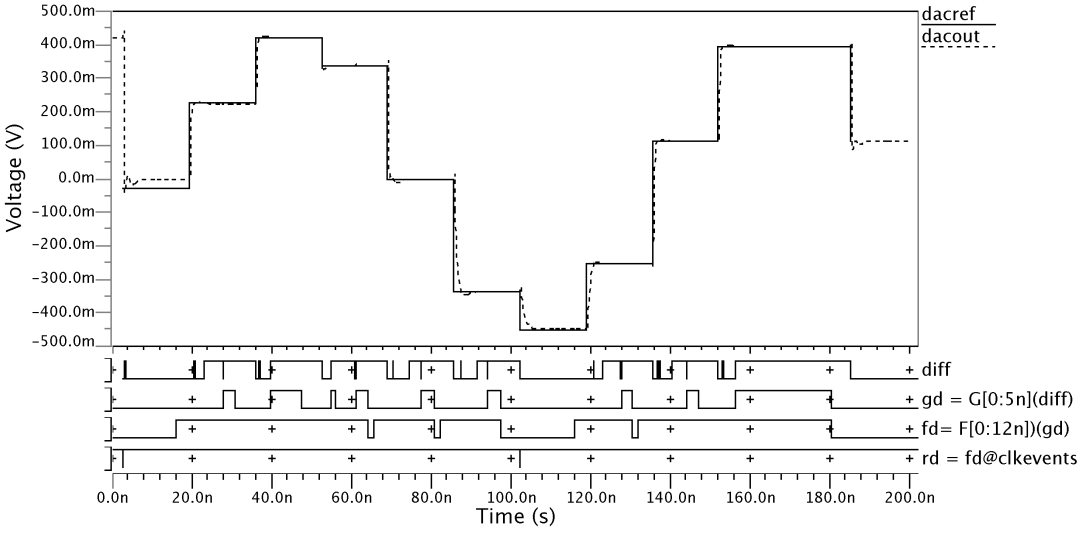


Figure 6.4. Evaluation steps of Assertion 6.9.

6.3. Programmable Gm-C Filter

We investigate a digitally-programmable continuous-time Gm-C low-pass filter design to integrate circuit analyses in our assertion-based verification flow in this case study. In a typical receiver system used in communication applications, three basic operations are performed: amplification, filtering and data conversion. Programmable gain amplifiers (PGAs), low-pass filters (LPFs), analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) are designed to implement these operations in actual circuits. In next-generation applications, these circuits should be very flexible and capable of adapting their performance to different standard requirements with reduced power consumption [44]. To achieve this, we need to design and verify analog circuits with digital control of biasing, gain, circuit- and stage-level reconfiguration, block power down/up.

Designers can implement filter designs in both digital and analog domains. Domain selection includes many design trade-offs and challenges. Although filtering at digital domain is preferred over analog domain, because of digital domain's robustness and scalability, the overwhelming requirements for following data conversion step make programmable analog filters attractive for new generation mixed-signal applications. Among two common programmable analog filter methods, active-RC and Gm-C circuit techniques, Gm-C technique is preferred for high-frequency and low-power applications.

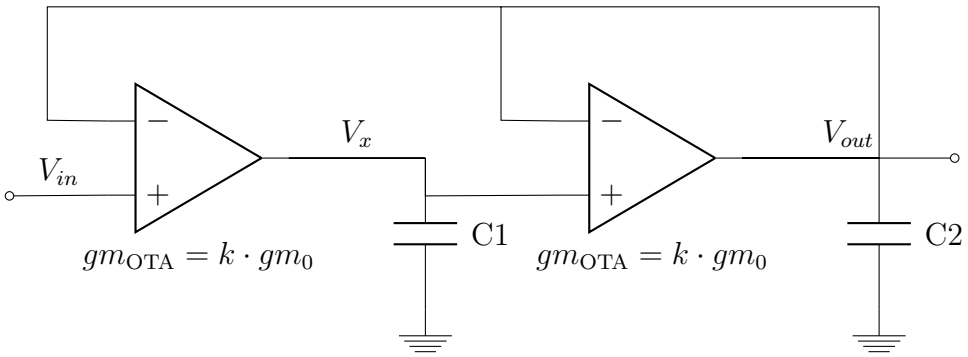


Figure 6.5. The programmable low-pass filter circuit.

To demonstrate our approach, we have designed a digitally-programmable low-pass Gm-C filter. Its schematic can be seen in Figure 6.5, where amplifier blocks in the circuit are classified as operational transconductance amplifier (OTA). Cut-off frequency for this circuit is determined using the following equation:

$$f_c = \frac{K \cdot gm_0}{2\pi\sqrt{C_1 \cdot C_2}} \tag{6.1}$$

where f_c denotes cut-off frequency of the filter, g_m denotes transconductance value of amplifiers, C_1, C_2 denote capacitor values in the circuit and K is the decimal value of 3-bit binary input, which provides programmability for this filter. We can change cut-off frequency of the low-pass filter during operation by adjusting K. We implemented this circuit using 0.18um technology and used a SPICE simulator for circuit analyses.

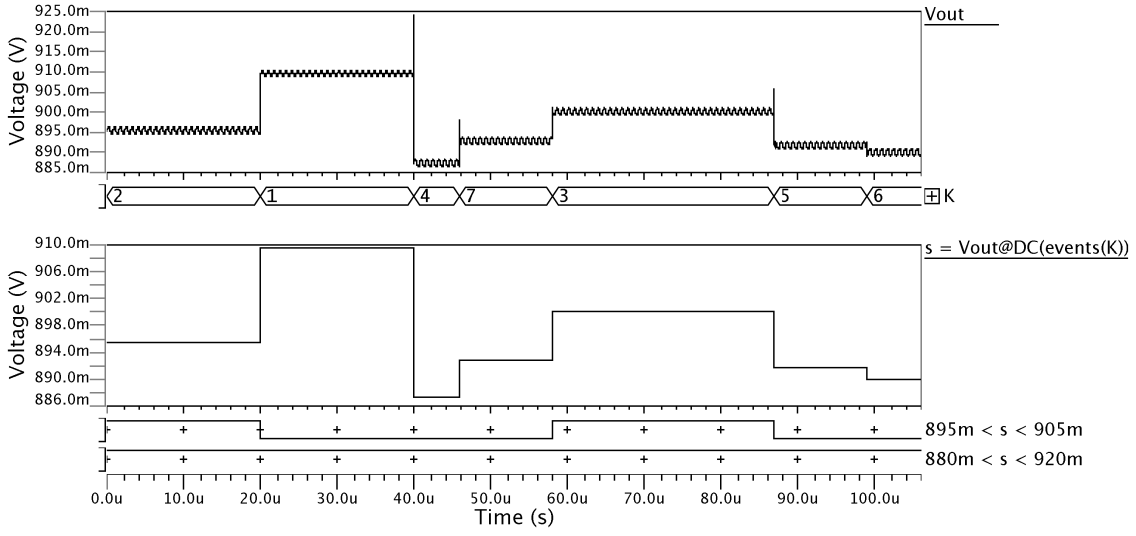


Figure 6.6. Monitoring time-varying DC level of the filter output according to K.

In verification of programmable analog filters, we should check DC operating

points of different states. The simulator performs a DC analysis and computes DC levels of all nodes. Then, we check if these values satisfy desired conditions. However, changing 3-bit digital K value can disrupt DC levels in the circuit and cause a shift in DC level of the output node. On the other hand, keeping DC level in a specified range is important for a robust system. Therefore, the simulator performs a DC analysis whenever a change in K occurs and we check DC levels for each change during simulation. In Figure 6.6, we illustrate the change in DC level of the output node based on time and varying K. If we want to keep the value of output DC level of the filter between 880mV and 920mV, we write this property:

Assertion

$$Assert: \qquad \mathcal{G}\left(880mV < (Vout)@DC(e_k) < 920mV\right) \tag{6.10}$$

end

where e_K denotes an event for a change in K determined by digital inputs. In Figure 6.6, we can see that varying DC level for output node is always between specified values so the assertion is satisfied. However, a stricter requirement such as [895m, 905m] would fail for this circuit. We annotate the results of these checks in the form of boolean signals in Figure 6.6.

As next step, we verify time-varying AC characteristics of the our low-pas filter. Design specification states that bandwidth value of the low-pass filter should be changed linearly depending on the value of K factor. According to Equation 6.1, bandwidth value of the filter should be approximately equal to $2.1 \cdot k$ MHz (eg. 2.1 MHz if k=1, 8.4 MHz if k=4) in the ideal case if we select circuit parameters, gm_0, C_1 and C_2 as 25μA/V, 1pF and 4pF, respectively. To verify linearity of bandwidth adjustment, one approach we can take is to extract *bandwidth per K* metric and check whether it always stays in acceptable region. This way we ensure that the error deviated from expected value stay in specified limits. This approach is captured in Assertion 6.11.

Assertion

$$Assert: \qquad \mathcal{G}\left\{2.0MHz < \Big(BW(\ Vout\ )@AC(e_K)\Big)/K < 2.2MHz\right\} \qquad (6.11)$$

end

In Assertion 6.11, for each change in K, an AC analysis is performed and a frequency plot is returned. Bandwidth operator (BW) computes bandwidth value from each AC analysis, and we plot bandwidth of the filter versus time in Figure 6.7. We can see that Assertion 6.11 is evaluated true because *bandwidth per K* metric always remain inside $[2M, 2.2M]$ region.

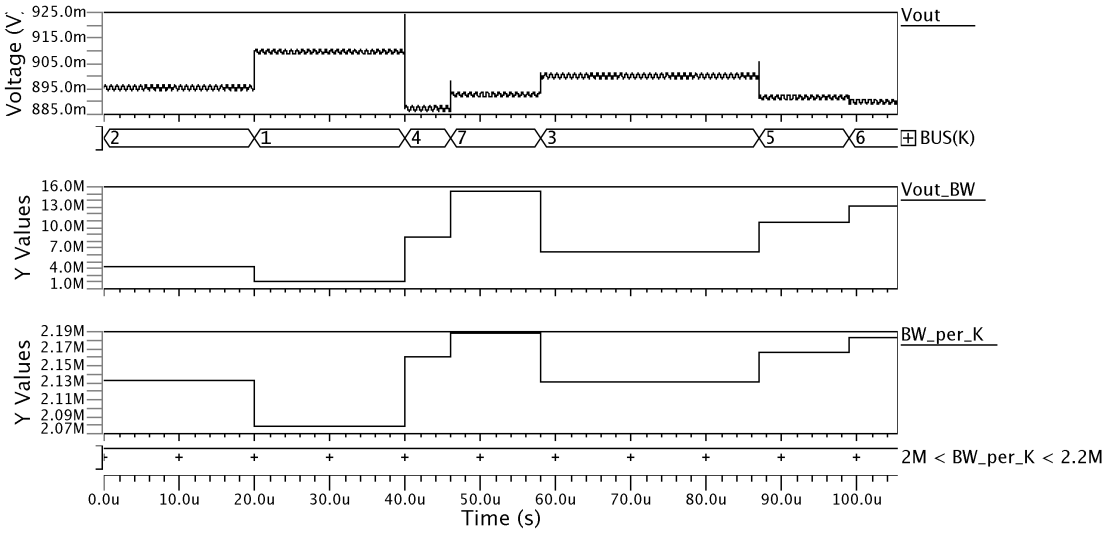


Figure 6.7. Monitoring time-varying cutoff frequency value of the filter according to K factor.

Although Assertion 6.11 is true for this case, we experiment alternative approach to check linearity of bandwidth adjustment. This approach aims to use linear regression algorithm inside in our assertion-based verification flow. Basically, we monitor bandwidth value of the filter, fit a line on monitored values and check the goodness of fitted line. We captured this approach in Assertion 6.12.

Assertion

$$Assert: \qquad \mathcal{G}(r > 0.999) \tag{6.12}$$

where,

$$r = linregres(K, BW(\ Vout\ )@AC(e_K))$$

end

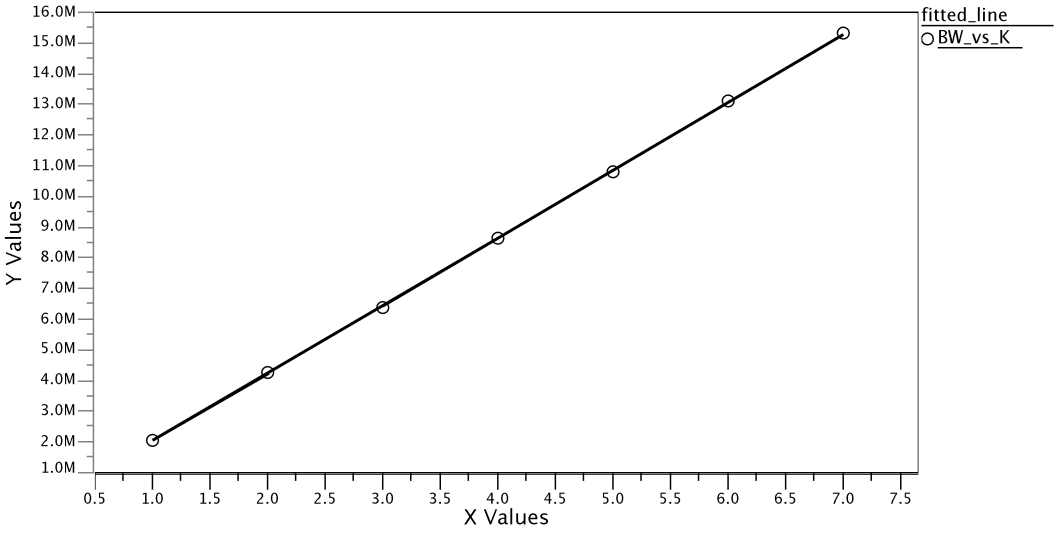


Figure 6.8. Bandwidth vs K and fitted line.

In Figure 6.8, we plot bandwidth versus K and fitted line for illustration. The linear regression operator *linregres* returns correlation coefficient r, which shows goodness of fit. For a perfect fit, $r = 1$ so values close to 1 means a better linearity. In Assertion 6.12, we check whether r value is greater than 0.999. *linregress* operator returns 0.9994 for this dataset so this assertion is satisfied in this case. Linear regression example shows that scientific data analysis algorithms have crucial importance for analog verification, and unified and extensible environment increases productivity as well as verification quality.

Ultimately, a filter is designed to implement a linear operation however actual implementations of filter circuits (or any other circuit that implements a linear operation) are slightly nonlinear because of nonlinearity of transistor devices. FFT-based THD and SNDR metrics are used to measure the amount of nonlinearity of designs from transient simulation results.

We captured THD and SNDR specifications in Assertion 6.13 and Assertion 6.14.

Assertion

$$Assert: \qquad \mathcal{G}\left\{0.0 < THD(\ Vout\ )@FFT(e_K) < 3.0\right\} \tag{6.13}$$

end

Assertion

$$Assert: \qquad \mathcal{G}\left\{SNDR(\ Vout\ )@FFT(e_K) > 30.0\right\} \tag{6.14}$$

end

In Figure 6.9, we plot the output node of the filter and monitor these properties for our filter design according to the change in K and we annotate THD and SNDR calculation on the plot. We see that THD and SNDR specifications are not satisfied for all K. Unsatisfied linearity specifications usually do not have easy fixes and may require a change in circuit topology. However, the same assertions can be used for several topologies; therefore, verification effort across different topologies is reduced compared to manual verification.

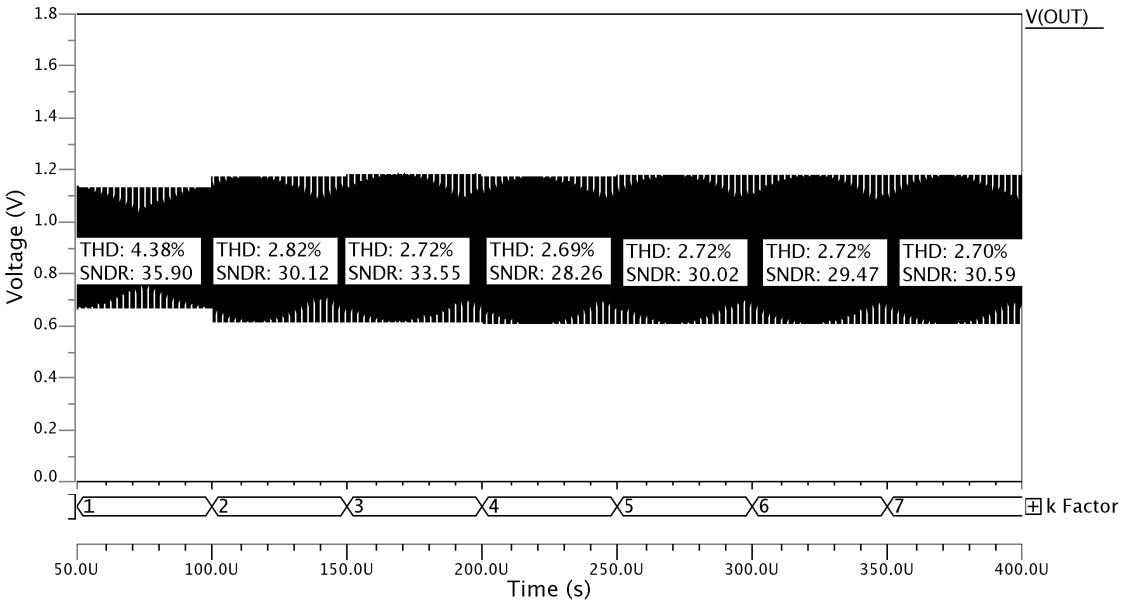


Figure 6.9. Monitoring time-varying THD and SNDR values of the filter according to K.

7. CONCLUSION

Traditional simulation-based AMS verification methodology lacks formality, reusability and structuredness. Formal methods that prove 100% correctness of AMS designs suffer from state explosion problem and they become infeasible quickly as the size of AMS design increases. Assertion based verification methodology for AMS verification provides the middle ground between simulation-based and formal methods by formalizing property specification and evaluation as well as being practical, straightforward and scalable.

Assertion based verification flow adapted from digital verification technology extends discrete system verification techniques towards continuous and hybrid systems domain. However, AMS properties have very diverse nature, many different analysis techniques ranging from signal processing algorithms to statistical techniques. Therefore, AMS assertion languages would be very limited for AMS verification community if they do not have expressive constructs to handle all these aspects of AMS design. In this thesis, we aimed to enrich analog expressiveness of AMS assertion languages by considering this analog nature, as well as formalizing and integrating conventional AMS verification into assertion based verification flow.

We studied analog tolerances and comparison problem, which is used in deciding if two analog waveforms are equivalent in Chapter 3. Determining tolerances around analog signals affects directly the quality of verification and the result of assertions. Therefore, tolerance management and smart comparison algorithms should be an integral part in AMS verification environments. Conventional AMS verification techniques such as measurements and circuit analyses are well-defined techniques to collect information about AMS designs. These analysis techniques would be needed for complete AMS verification. Therefore, we integrated these analysis techniques into assertion based AMS verification by using events in Chapter 4.

We developed AMS-Verify framework to implement our proposals and performed

three case studies. Case studies reveals that AMS verification needs flexible property specification constructs such as defining parametric assertions, local variables and loops without sacrificing formality. Signal processing and statistical techniques would be more important for AMS assertions languages in the future therefore providing constructs to support these techniques is a step forward for expressive AMS assertions. AMS-Verify framework is based on Python language, and Python has flexible and extensible nature and offers excellent scientific libraries, which is very useful for AMS verification.

For future, tolerances for analog signals and analog comparison can be extended towards more formal approaches such as reachability analysis and even better supporting statistical specifications. In many cases to prevent overdesign and/or relaxing other specifications, ensuring 95% correctness instead of ensuring 100% correctness may be more desirable for AMS verification. Developing assertion languages allowing to write such statistical specifications is an open research area. Similarly, techniques to measure coverage information using assertions should be developed for AMS designs. Further, although cooperation between simulator and assertion checker can be realized for simple assertions, it is not possible for assertions including circuit analyses and measurements. Current AMS simulators do not support true event-driven circuit analyses and measurements and analysis times should be explicitly defined before simulation. Improving this situation would be beneficial to achieve true event-driven assertion based verification. Finally, we believe that case studies are very important for development of AMS assertion languages and different real world AMS designs should be investigated.

APPENDIX A: AMS-VERIFY TUTORIAL

AMS-Verify is an analog and mixed-signal (AMS) verification framework inspired from unit testing methodology in software verification. AMS-Verify framework provides common routines and analyses which extensively used in AMS verification. The structure of AMS-Verify framework is similar to *unittest* module of standard Python library, which has a rich heritage as part of the xUnit family of testing libraries.

A.1. Writing Assertion Suites

Verification of AMS designs using AMS-Verify starts with writing assertions capturing desired AMS properties. In AMS-Verify framework, assertions are implemented as Python methods and assertion are collected in assertion suites. To write assertion suites, first import `AssertionSuite` module as follows.

```
from amsverify.assertionsuite import AssertionSuite
```

Then, create an assertion suite by subclassing `AssertionSuite`. Methods whose name starts with "verify" word are identified as AMS assertions. Simplest assertion suite includes only one assertion as follows.

```
class User_Created_Assertion_Suite(AssertionSuite):
    verify(self):
        self.assert_true(True)
```

`assert*` methods provided by AMS-Verify framework can be used to assert desired AMS properties. AMS-Verify support assertions of temporal properties in AMS designs. Therefore, four `assert*` methodsThis allows us to run all of the test code just

by running the file.This allows us to run all of the test code just by running the file. are available in AMS-Verify framework.

- `assert_true`
- `assert_false`
- `assert_always`
- `assert_eventually`

AMS assertions usually need to set up verification environment such as reading simulated waveforms from database or calculating design constants. Setup procedures can be common for more than one assertion therefore `setUp` and `tearDown` methods are provided to prevent code repeating in assertions. If you define `setUp` method in an assertion suite, it will be called before every assertion. Similarly, if you define `tearDown` method in an assertion suite, it will be called after every assertion. For example, `setUp` and `tearDown` methods are called independently for each assertion in the assertion suite below.

```
class Properties_ABC(AssertionSuite):
    setUp(self):
        # calls before verify_A, verify_B, verify_C
    verify_A(self):
        # Assertion A
    verify_B(self):
        # Assertion B
    verify_C(self):
        # Assertion C
    tearDown(self):
        # calls after verify_A, verify_B, verify_C
```

A.2. Running Assertion Suites

Assertions are verified by running assertion suites for each assertion separately. The easiest way to run AMS-Verify assertions is to include:

```
if __name__ == "__main__":
    User_Created_Assertion_Suite().run("verify")
```

at the bottom of each test file, then simply run the script directly from the command line:

```
python user_created_assertion_suite.py
```

Assertion checking returns a True/False answer together with elapsed CPU time info as below:

```
>>> verify ... True (Elapsed time: 0.011324989112 s)
```

APPENDIX B: ASSERTION SOURCE FILES

```
from amsverify.assertionsuite import AssertionSuite

class SettlingTimeProperty(AssertionSuite):
    def setUp(self):
        self.dataset_open('settling.wdb')
        self.wf0 = self.wf('<settling>signal_a', alias='signal_a')

    def verify(self):
        def prop_stability(wf, settlingtime, settleduration, offset, tol, alias):
            i = self.inrange(wf, offset+tol, offset-tol, alias=alias+"_ib")
            a = self.always(i, ubound=settleduration, lbound=0, alias=alias+"_ab")
            e = self.eventually(a, ubound=settlingtime, lbound=0, alias=alias+"_b")
            return e
        redge = prop_stability(self.wf0, 1.115e-9, 0.5e-9, 1.8, 0.01, alias='redge')
        fedge = prop_stability(self.wf0, 1.115e-9, 0.5e-9, 0.0, 0.01, alias='fedge')
        rise_events = self.crossing(self.wf0, ylevel=0.9, slopetrigger="rising")
        fall_events = self.crossing(self.wf0, ylevel=0.9, slopetrigger="falling")
        k1 = self.at_event(redge, events=rise_events, alias='k1b')
        k2 = self.at_event(fedge, events=fall_events, alias='k2b')
        p = self.and2(k1, k2, alias='pb')
        self.assert_always(p)

    def tearDown(self):
        self.dataset_save()

if __name__ == '__main__':
    SettlingTimeProperty().run('verify')
```

Figure B.1. Source file of Settling Time Property in Assertion 6.8.

```
from amsverify.assertionsuite import AssertionSuite

class SHDAC(AssertionSuite):
    def setUp(self):
        self.dataset_open('shdac.wdb')
        self.dacclk = self.wf(
            "<shdac/test_all7_eldonet/TRAN>V(FLOP_L1_CLK)",
            alias='dacclk')
        self.sh_out = self.wf(
            "<shdac/test_all7_eldonet/TRAN>V(X_10BIT_FLASH_ADC1.SH)",
            alias='sh_out')
        self.dacout = self.wf(
            "<shdac/test_all7_eldonet/TRAN>V(X_10BIT_FLASH_ADC1.DACOUT)",
            alias='dacout')
    def verify(self):
        hold_times = self.crossing(self.dacclk, ylevel=-0.899, slopetrigger="rising")
        hold_events = [dict(t=e['t'],
            yval=int((self.yval(self.sh_out, e['t'])+0.45)/0.028125)*0.028125-0.45)
            for e in hold_times]
        dacref = self.datatowf(
            [(e['t'], e['yval']) for e in hold_events], staircase=True, alias="dacref")
        bdiff = self.compare(dac, dacref, 'absolute', tol='1e-3', alias="bdiff")
        gbdiff = self.always(bdiff, ubound=5e-9, lbound=0, alias="G_bdiff")
        fgbdiff = self.eventually(gbdiff, ubound=12e-9, lbound=0, alias="F_G_bdiff")
        p = self.at_event(fgbdiff, events=hold_events, alias='p')
        self.assert_always(p)
    def tearDown(self):
        self.dataset_save()

if __name__ == '__main__':
    SHDAC().run('verify')
```

Figure B.2. Source file of SH and DAC property in Assertion 6.9.

```
from amsverify.assertionsuite import AssertionSuite

class FilterGMC_DC(AssertionSuite):

    def setUp(self):
        self.dataset_open('filter-1mA.wdb')
        self.analysis_points = [
            1e-6, 21e-6, 41e-6, 47e-6, 59e-6,
            88e-6, 100e-6]
        self.dc_events = [dict(t=p) for p in self.analysis_points]
        self.voutatdc = self.at_dc('VOUT', self.dc_events, alias='s')

    def verify(self):
        p = self.inrange(self.voutatdc, 0.920, 0.880, alias='p')
        self.assert_always(p)

    def tearDown(self):
        self.dataset_save()

if __name__ == '__main__':
    FilterGMC_DC().run('verify')
```

Figure B.3. Source file of DC voltage property of programmable filter in Assertion 6.10.

```
from amsverify.assertionsuite import AssertionSuite

class FilterGMC_AC(AssertionSuite):

    def setUp(self):

        self.dataset_open('filter-1mA.wdb')
        self.points = [
            1e-6, 21e-6, 41e-6, 47e-6, 59e-6,
            88e-6, 100e-6]
        self.k_values = [2, 1, 4, 7, 3, 5, 6]
        self.ksignal = self.wftodata(zip(self.points, self.k_values), staircase=True)
        self.ac_events = [dict(t=p) for p in self.points]
        self.bwatac = self.bandwidth('VOUT', self.ac_events, alias='s')

    def verify(self):

        K = self.wftodata(staircase=True)
        self.bwperk = self.bwatac / self.ksignal
        p = self.inrange(self.bwperk, 2.0e6, 2.2e6, alias='p')
        self.assert_always(p)

    def tearDown(self):

        self.dataset_save()

if __name__ == '__main__':
    FilterGMC_AC().run('verify')
```

Figure B.4. Source file of AC voltage property of programmable filter in Assertion 6.11.

```
import numpy as np
from scipy import stats
from amsverify.assertionsuite import AssertionSuite

class FilterGMC_LG(AssertionSuite):

    def setUp(self):

        self.dataset_open('filter-1mA.wdb')
        self.points = [
            1e-6, 21e-6, 41e-6, 47e-6, 59e-6,
            88e-6, 100e-6]
        self.k_values = [2, 1, 4, 7, 3, 5, 6]
        self.ksignal = self.wftodata(zip(self.points, self.k_values), staircase=True)
        self.ac_events = [dict(t=p) for p in self.points]
        self.bwatac = self.bandwidth('VOUT', self.ac_events, alias='s')

    def verify(self):

        xi = [e[1] for e in self.ksignal]
        yi = [e[1] for e in self.bwatac]

        slope, intercept, r_value, p_value, std_err = stats.linregress(xi, yi)

        p = self.gt(r_value, 0.999)

        self.assert_always(p)

    def tearDown(self):
        self.dataset_save()

if __name__ == '__main__':
    FilterGMC_LG().run('verify')
```

Figure B.5. Source file of regression property of programmable filter in Assertion 6.12.

```
from amsverify.assertionsuite import AssertionSuite

class FilterGMC_THD(AssertionSuite):

    def setUp(self):

        self.dataset_open('filter-100mA.wdb')

        self.points = [
            50e-6, 100e-6, 150e-6, 200e-6,
            250e-6, 300e-6, 350e-6]

        self.fft_events = [dict(t=p) for p in self.points]
        self.thdatfft = self.thd('VOUT', self.fft_events, alias='s')

    def verify(self):

        p = self.inrange(self.thdatfft, 3.0, 0.0, alias='p')

        self.assert_always(p)

    def tearDown(self):

        self.dataset_save()

if __name__ == '__main__':
    FilterGMC_THD().run('verify')
```

Figure B.6. Source file of THD property of programmable filter in Assertion 6.13.

```
from amsverify.assertionsuite import AssertionSuite

class FilterGMC_SNDR(AssertionSuite):

    def setUp(self):

        self.dataset_open('filter-100mA.wdb')

        self.points = [
            50e-6, 100e-6, 150e-6, 200e-6,
            250e-6, 300e-6, 350e-6]

        self.fft_events = [dict(t=p) for p in self.points]
        self.sndratfft = self.sndr('VOUT', self.fft_events, alias='s')

    def verify(self):

        p = self.gt(self.sndratfft, 30.0, alias='p')

        self.assert_always(p)

    def tearDown(self):

        self.dataset_save()

if __name__ == '__main__':
    FilterGMC_SNDR().run('verify')
```

Figure B.7. Source file of SNDR property of programmable filter in Assertion 6.14.

REFERENCES

1. Barke, E., D. Grabowski, H. Graeb, L. Hedrich, S. Heinen, R. Popp, S. Steinhorst and Y. Wang, "Formal Approaches to Analog Circuit Verification", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 724–729, 2009.

2. Zaki, M. H., S. Tahar and G. Bois, "Review: Formal Verification of Analog and Mixed Signal Designs: A Survey", *Microelectronics Journal*, pp. 1395–1404, 2008.

3. Alur, R., "Formal Verification of Hybrid Systems", *Proceedings of the Conference on Embedded Software (EMSOFT)*, pp. 273–278, 2011.

4. Hartong, W., L. Hedrich and E. Barke, "On Discrete Modeling and Model Checking for Nonlinear Analog Systems", *Proceedings of the Conference on Computer Aided Verification (CAV)*, pp. 401–413, 2002.

5. Grabowski, D., D. Platte, L. Hedrich and E. Barke, "Time Constrained Verification of Analog Circuits Using Model-Checking Algorithms", *Electronic Notes in Theoretical Computer Science (ENTCS)*, Vol. 153, No. 3, pp. 37–52, 2006.

6. Dastidar, T. R. and P. Chakrabarti, "A Verification System for Transient Response of Analog Circuits", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 12, No. 3, pp. 31:1–31:39, 2008.

7. Steinhorst, S. and L. Hedrich, "Model Checking of Analog Systems Using an Analog Specification language", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 324–329, 2008.

8. Al-Sammane, G., M. Zaki and S. Tahar, "A Symbolic Methodology for the Verification of Analog and Mixed Signal Designs", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 249–254, 2007.

9. Little, S., D. Walter, K. Jones, C. Myers and A. Sen, "Analog/Mixed-Signal Circuit Verification Using Models Generated from Simulation Traces", *International Journal of Foundations of Computer Science*, Vol. 21, No. 2, pp. 191–210, 2010.

10. Wang, Z., M. H. Zaki and S. Tahar, "Statistical Runtime Verification of Analog and Mixed Signal Designs", *Proceedings of the Conference on Signals, Circuits and Systems (SCS)*, pp. 1–6, 2009.

11. Clarke, E., A. Donzé and A. Legay, "On Simulation-Based Probabilistic Model Checking of Mixed-Analog Circuits", *Formal Methods in System Design*, Vol. 36, No. 2, pp. 97–113, 2010.

12. Tiwary, S. K., A. Gupta, J. R. Phillips, C. Pinello and R. Zlatanovici, "First Steps Towards SAT-Based Formal Analog Verification", *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2009.

13. Miller, M. and F. Brewer, "Formal Verification of Analog Circuit Parameters Across Variation Utilizing SAT", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 1442–1447, 2013.

14. Dang, T., A. Donzé and O. Maler, "Verification of Analog and Mixed-Signal Circuits Using Hybrid System Techniques", *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pp. 21–36, 2004.

15. Frehse, G., B. H. Krogh and R. A. Rutenbar, "Verifying Analog Oscillator Circuits Using Forward/Backward Abstraction Refinement", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 257–262, 2006.

16. Frehse, G., C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang and O. Maler, "SpaceEx: Scalable Verification of Hybrid Systems", *Proceedings of the Conference on Computer Aided Verification (CAV)*, pp. 379–395, 2011.

17. Steinhorst, S. and L. Hedrich, "Advanced Methods for Equivalence Checking of Analog Circuits with Strong Nonlinearities", *Formal Methods in System Design*, Vol. 36, No. 2, pp. 131–147, 2010.

18. Singh, A. and P. Li, "On Behavioral Model Equivalence Checking for Large Analog/Mixed Signal Systems", *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 55–61, 2010.

19. Steinhorst, S. and L. Hedrich, "Equivalence Checking of Nonlinear Analog Circuits for Hierarchical AMS System Verification", *Proceedings of the Conference on VLSI and System-on-Chip (VLSI-SoC)*, pp. 135–140, 2012.

20. Maler, O. and D. Ničković, "Monitoring Temporal Properties of Continuous Signals", *Proceedings of the Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, pp. 152–166, 2004.

21. Thati, P. and G. Roşu, "Monitoring Algorithms for Metric Temporal Logic Specifications", *Electronic Notes in Theoretical Computer Science (ENTCS)*, Vol. 113, pp. 145–162, 2005.

22. Fainekos, G. E., A. Girard and G. J. Pappas, "Temporal Logic Verification Using Simulation", *Proceedings of the Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, pp. 171–186, 2006.

23. Fainekos, G. E. and G. J. Pappas, "Robustness of Temporal Logic Specifications for Continuous-Time Signals", *Theoretical Computer Science (TCS)*, Vol. 410, No. 42, pp. 4262–4291, 2009.

24. Mukhopadhyay, R., S. K. Panda, P. Dasgupta and J. Gough, "Instrumenting AMS Assertion Verification on Commercial Platforms", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 14, No. 2, pp. 21:1–21:47, 2009.

25. Lämmermann, S., J. Ruf, T. Kropf, W. Rosenstiel, A. Viehl, A. Jesser and

L. Hedrich, "Towards Assertion-Based Verification of Heterogeneous System Designs", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 1171–1176, 2010.

26. Ma, M., L. Hedrich and C. Sporrer, "ASDeX: A Formal Specification for Analog Circuit Enabling a Full Automated Design Validation", *Design Automation for Embedded Systems*, pp. 1–20, 2012.

27. Maler, O., D. Ničković and A. Pnueli, "Checking Temporal Properties of Discrete, Timed and Continuous Behaviors", *Pillars of computer science*, pp. 475–505, 2008.

28. Maler, O. and D. Ničković, "Monitoring Properties of Analog and Mixed-Signal Circuits", *International Journal on Software Tools for Technology Transfer (STTT)*, pp. 1–22, 2013.

29. Mukherjee, S., P. Dasgupta, S. Mukhopadhyay, S. Little, J. Havlicek and S. Chandrasekaran, "Synchronizing AMS Assertions with AMS Simulation: From Theory to Practice", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 17, No. 4, pp. 38:1–38:25, 2012.

30. Donze, A., O. Maler, E. Bartocci, D. Nickovic, R. Grosu and S. Smolka, "On Temporal Logic and Signal Processing", *Proceedings of the Symposium on Automated Technology for Verification and Analysis (ATVA)*, pp. 92–106, 2012.

31. Ulus, D. and A. Sen, "Analog ve Karışık İşaret Gözcü Tabanlı Doğrulamada Halelerin Kullanımı", *Gömülü Sistemler ve Uygulamaları (GÖMSİS) Sempozyumu Bildiri Kitabı*, 2012.

32. Ulus, D. and A. Sen, "Using Haloes in Mixed-Signal Assertion-Based Verification", *Proceedings of the Workshop on High Level Design Validation and Test (HLDVT)*, pp. 49–55, 2012.

33. Ulus, D., A. Sen and F. Baskaya, "Analog Layer Extensions for Analog/Mixed-

Signal Assertion Languages", *Proceedings of the Conference on Very Large Scale Integration (VLSI-SoC)*, 2013.

34. Ulus, D., A. Sen and F. Baskaya, "Integrating Circuit Analyses for Assertion-Based Verification of Programmable AMS Circuits", *Proceedings of the Forum on Specification and Design Languages (FDL)*, 2013.

35. Pnueli, A., "The Temporal Logic of Programs", *Proceedings of the Symposium on Foundations of Computer Science*, pp. 46–57, 1977.

36. Clarke, E. M. and E. A. Emerson, *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic*, Springer, 1982.

37. Koymans, R., "Specifying Real-Time Properties with Metric Temporal Logic", *Real-time systems*, Vol. 2, No. 4, pp. 255–299, 1990.

38. Alur, R., T. Feder and T. Henzinger, "The Benefits of Relaxing Punctuality", *Journal of the ACM (JACM)*, Vol. 43, No. 1, pp. 116–146, 1996.

39. Maler, O., D. Ničković and A. Pnueli, "Real Time Temporal Logic: Past, Present, Future", *Proceedings of the Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, pp. 2–16, 2005.

40. Foster, H., E. Marschner and Y. Wolfsthal, "IEEE 1850 PSL: The Next Generation", *Proceedings of Design and Verification Conference and exhibition (DVCON)*, 2005.

41. Mentor Graphics Corporation, *EZwave User's and Reference Manual*, 2012.

42. Oliphant, T. E., "Python for Scientific Computing", *Computing in Science & Engineering*, Vol. 9, No. 3, pp. 10–20, 2007.

43. Esen, V. B., *10-Bit 60 MS/s Two-Step Flash ADC Design*, Master's Thesis, Bogazici University, 2013.

44. de la Rosa, J. M., R. Castro-Lopez, A. Morgado, E. C. Becerra-Alvarez, R. del Rio, F. V. Fernandez and B. Perez-Verdu, "Adaptive CMOS Analog Circuits for 4G Mobile Terminals –Review and State-of-the-Art Survey", *Microelectronics Journal*, Vol. 40, No. 1, pp. 156 – 176, 2009.