Arij BEN JMAA

MASTER THESIS REPORT
2019-2020

MELEXIS
Chemin de Buchaux 38, 2022 Bevaix, Switzerland

Application of SystemC AMS in the verification of mixed signal systems

from 02/03/2020 to 31/08/2020

Confidentiality: No

Under the supervision of:

- **Company supervisor: Alessandro BASILI, abs@melexis.com**
  Present at the defense: Yes
- **EPFL Tutor: Alain VACHOUX, alain.vachoux@epfl.ch**
- **Phelma Tutor: Michele PORTOLAN, michele.portolan@grenoble-inp.fr**

Ecole nationale supérieure de physique, électronique, matériaux

Phelma
Bât. Grenoble INP - Minatec
3 Parvis Louis Néel - CS 50257
F-38016 Grenoble Cedex 01

Tél +33 (0)4 56 52 91 00
Fax +33 (0)4 56 52 91 03

http://phelma.grenoble-inp.fr

Table of Contents

Glossary

AMS: Analog Mixed Signal

LSB: Least Significant Bit

MSB: Most Significant Bit

TDF: Timed Data Flow

LSF: Linear Signal Flow

ELN: Electrical Linear Networks

PLL: Phase Locked Loop

VCO: Voltage Controlled Oscillator

PHC: Phase Comparator

LPF: Loop Filter

DAC: Digital to Analog Converter

MUX: Multiplexer

DNL: Differential Nonlinearity

INL: Integral Nonlinearity

List of figures

1 INTRODUCTION

1.1 Presentation of the company

Melexis is a semiconductor manufacturing company. It was founded in Belgium in 1988 and today it includes over 1500 employees in 19 sites located in 3 different continents [4]. The Project took place at the corporate site of Bevaix, Switzerland.

Melexis designs, develops and produces micro-electronic devices based on integrated circuits that combine analog and digital signals, generally dedicated to automotive applications. Nevertheless, Melexis manufactures also advanced systems for other application fields among them we can list the smart buildings, the industry, the transportation and the medical domain [4]. Melexis products can be classified into 3 different categories [3]:

- Sensors: Among them we can list the position sensors ICs, the latch and switch ICs, the current sensor ICs, the temperature sensor ICs and the optical sensor ICs [3].
- Drivers: Among them we can list the embedded motor driver ICs, the fan driver ICs, the LED driver ICs and the PRE-driver ICs [3].
- Communicate ICs: like the radio frequency receiver ICs and the radio frequency transmitter ICs [3].

The majority of the work was carried out unfortunately from home because of the occasional circumstances of the epidemic. Although the work was not done on site, I had the opportunity to meet the Melexis team and exchange with them during the welcoming day and during the regular meetings we had by video conferences. They are very helpful and comprehensive persons and the atmosphere seems to be friendly and encouraging for work. I found myself very welcomed and people were very happy that I joined the team with the rest of the trainees.

1.2 Presentation of the project context

Several methodologies already exist for the verification of the functional behavior of digital systems providing credible results. These methods ensure the functional correctness of digital designs. Nonetheless, it is not often the case for analog/mixed signal ICs due to the complexity of their design in comparison with only-analog and only-digital signal ICs [9].

Over the years, it has been proved that merging digital and analog integrated circuit blocks in the same chips is a very beneficial procedure and provides many advantages among them we can enumerate the reduction of cost, the better reliability and the lower energy consumption [10]. This leads us to seek to find reliable methodologies of verification flow for analog/mixed signal systems.

1.3 Problematic

The current verification flow of the digital and analog/mixed signal designs used by the majority of the semiconducting companies is based on System Verilog, Verilog AMS and UVM. As an example, the analog/mixed signal ICs team in Melexis uses event driven model

to simulate the behavior of their designs which is better than solving complicated equations consuming time and energy. However, the high number of generated events still limit the simulation performances. Therefore, Melexis is looking for more efficient solutions that consume less time and energy.

1.4 Motivations

Since SystemC AMS supports different modeling and simulation techniques and offers discrete-time modeling based on the TDF (Timed Data Flow) formalism, developing a verification environment founded on the SystemC AMS language is potentially an interesting alternative.

Here comes the idea to define, up to where the Verilog AMS methodologies already set by Melexis can be replaced by SystemC AMS models and to explore the capabilities of such language for developing models of analog and mixed signal systems.

1.5 Objectives

The objectives of this internship can be summarized in these two following points:

- To develop a practical knowledge on developing SystemC and SystemC AMS models of analog and mixed signal components. This includes defining modeling and simulation guidelines.
- To set-up a working SystemC AMS modeling and simulation environment. First as a stand-alone environment using open-source tools (Eclipse). Then, using the Cadence tool environment available in Melexis.

On a practical level, a training on the SystemC and the SystemC AMS was carried out at the beginning of this internship. For this purpose, some analog and mixed signal systems were studied and modeled such as: sinusoidal source, filters, phase comparator, VCO and PLL. After mastering the SystemC and SystemC AMS languages, I went on to understand the electronic circuits provided by Melexis and try to model them. I focused on the DAC of the Triaxis Analog backend of a HALL effect sensor to study its different operating modes and some if its non-idealities. This work was first done and simulated in a simple environment (Eclipse). The second step is to install a more complex environment (Cadence) and try to simulate the models already developed in this environment.

Before my internship, Melexis lacked knowledge about SystemC and SystemC AMS. All the analog and mixed signal models were based on the Verilog AMS. As a result of my work, the team got reference models that correctly describe the required ideal and non-ideal DAC functionalities and efficient time simulations that can be compared later to the Verilog AMS simulations. This work shows also the good practice of using SystemC and SystemC AMS and which works correctly in the two environments (Eclipse and Cadence).

2 INTRODUCTION TO SYSTEMC AMS

2.1 SystemC AMS motivations

The concept of combining embedded Hardware/Software systems with their analog physical environment is experiencing a significant growth [12]. Therefore, designing digital Hardware/Software systems that interwoven with analog and mixed signal ICs becomes one of today challenges. In this context, we can mention the sensors, the actuators, the RF interfaces and the power electronics [1] [12].

SystemC provides a discrete-event simulation enabling the functional verification of digital Hardware/Software systems. Nevertheless, providing simulation that reflects the behavior of continuous time analog systems is limited as it demands a description compatible with event-driven simulation. [1]

The SystemC AMS is an extension of the SystemC allowing a refinement methodology to model, design and simulate Embedded Analog/Mixed signals systems. These models meet the needs of the automotive and the semiconductors industries [1].

2.2 Model Abstractions

One of the advantages offered by SystemC AMS is the availability of many abstraction levels in addition to those supported by SystemC. These new abstraction models are the solution for the modeling and the simulation of the Embedded AMS systems [1]. For example, SystemC AMS distinguishes the discrete-time behavior from the continuous-time behavior on the one hand, and the conservative behavior from the non-conservative behavior on the other hand. In the next paragraph, I am going to focus on the first characteristic and present in more details the differences between the two behaviors (continuous-time and discrete-time) as it fits well the topic of my master thesis.

The discrete-time modeling is based on the abstraction of signals and physical quantities as values sampled in time using a fixed step sampling schema. These values could be either real or discrete (Boolean, integer…). Values between two successive time points are therefore not defined and assumed constant. Some continuous-time signals can be sampled and described by a discrete-time approach with the condition of having reasonable approximations [1].

The continuous-time behavior is more precise. Signals and physical quantities are modeled as real-valued functions of time. This approach is based on the resolution of mathematical equations which requires complex algorithms. Thus, it is obvious that simulating systems modeled as continuous-time behaviors will consume more energy and time than others modeled as discrete-time behaviors but the resolution will be better [1].

2.3 Modeling Formalism

SystemC AMS offers three different modeling formalism to guarantee high and different abstraction levels: Timed Data Flow (TDF), Linear Signal Flow (LSF), and Electrical

Linear Networks (ELN). In this section, I will detail the particularities of each computation model.

2.3.1 Timed Data flow

This formalism is based on discrete time modeling and not on the event driven modeling imposed by the SystemC. A specific scheduling of the TDF models is defined at the elaboration and before the simulation starts. The processing functions of the different TDF modules are then executed according to the direction of the dataflow. The signals which may be of any C++ type, propagate through the channels and ports that connect the different dataflow modules [1] (see section 2.4 for more details).

2.3.2 Linear Signal Flow

This formalism, in contrast to the previous one, offers the possibility of modeling continuous-time behaviors. This is done thanks to primitive modules that already exist in the SystemC AMS library among them we can list the addition, the multiplication and the integration. An LSF model is a system of linear equations that describes the behaviors of real valued signals in time domain and which have to be solved by a linear solver [1].

2.3.3 Electrical Linear Network

This Modeling formalism is based on instantiating predefined linear network primitives such as resistors and capacitors. They can after that be used as models to describe continuous-time behaviors [1].

2.4 Timed Data Flow modeling

Among the three different SystemC AMS modeling formalism, the TDF formalism will be considered in the rest of the document. It is the simplest formalism, the less energy and time consuming but also efficient for the modeling of the chosen component, the DAC. As mentioned before, this modeling formalism is based on discrete-time computation that considers the physical quantities as signals sampled in time. These signals carry discrete values over time and define quantities such us current and voltages [1]. To simplify things, let's consider the following diagram shown in Figure 1:

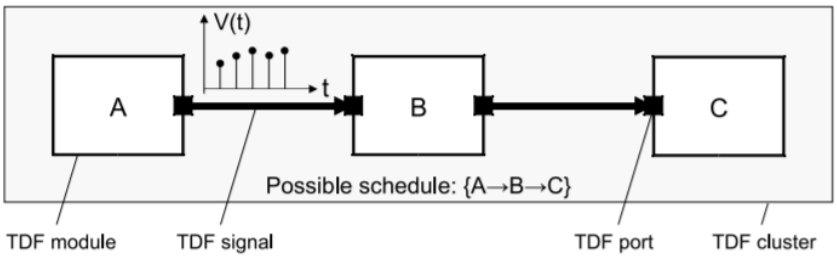

Figure 1: A basic TDF cluster with 3 TDF modules and 2 TDF signals [1]

This diagram presents a TDF cluster which is made of three TDF modules A, B and C connected to each other in a specific order. Each TDF module can be formed of many others

in turn as well. The TDF signals are the links between any modules and they form the connections. In other words, the modules form the vertices of the graph and the signals correspond to the edges [1]. TDF ports represent the inputs and outputs of the modules. A has only output ports, therefore it is the source. While C has only input ports, it is then the sink. Each module defines a processing function (can be mathematical) that acts on variables and signals and which is expressed in C++. The input and output signals are sampled in time. When the simulation starts, the cluster function which is the composition of the three functions of modules A, B and C is executed many times until the time of the simulation is over. The interval between two samples is called time step and is fixed by the user. We mention that the assignments of each module should be compatible with the static schedule already set (in this example the schedule is A→ B→ C). A cluster of TDF modules processes signals by repetitively activating the processing functions of the contained modules in the order of the static schedule. The step time should be therefore also compatible with the necessary time to finish the execution of the modules' functions [1].

In this work, only single rate processing, i.e., consuming one input sample and producing one output sample is considered. Nevertheless, the TDF formalism affords the possibility to set up delays and different rates if desired. For more details, check [1].

In this section, I will also present the two functions that I have used the most in my models, which are:

- Initialize function () which is an optional function and is used to initialize some data members such as private variables and the initial samples of ports [1]. This function is called only one time, at the end of elaboration and before the time simulation starts.
- Processing function () which is a mandatory function for the module definition and the time simulation. This function includes all the instructions that will act on the signals and the private variables. This function is called repetitively when the simulation starts each timepoint until the simulation time is over.

2.5 Examples

As an exemple of the TDF formalism in SystemC AMS, I modeled the behavior of an analog phase locked loop (PLL) and simulate it in three different cases.

A PLL is composed of three components: a phase comparator (PHC), a loop filter (LPF) and a voltage-controlled oscillator (VCO).

The PHC has two inputs signals, the reference signal $v_{ref}(t)$ and the VCO output signal v_{VCO}(t). The PHC output signal $v_{PHC}(t)$ is connected to the input port of the LPF. The output signal of the LPF $v_{ctrl}(t)$ is communicated to the VCO module input port. The PLL ensures that the $v_{ref}(t)$ and the $v_{VCO}(t)$ have the same frequency and a constant phase difference.

For this purpose, different models were developed:

- Phase comparator as a four-quadrant multiplier: it generates an output signal proportional to the phase difference between the two input signals as detailed in the following expression:

$$v_{PHC}(t)K.v_{ref}(t).v_{VCO}(t) \quad (1)$$

- K is the phase comparator gain.
  The PHC input signals are assumed to be sinusoidal sources as shown in the following expressions:

$$v_{ref}(t) = A_{ref}\ sin(\omega t) \quad (2)$$

$$v_{VCO}(t) = A_{VCO}\ sin(\omega t + \Phi_{VCO}) \quad (3)$$

- Loop filter: the filter parameters are the DC gain H_0 and the cut-off frequency f_p.
- Voltage controlled oscillator: it generates an output voltage $v_{VCO}(t)$ whose frequency f_{VCO} is proportional to the controlling input voltage $v_{ctrl}(t)$ as it can be observed in the following expression:

$$f_{VCO}(t) = f_{c0} + K_{VCO}.(v_{ctrl}(t) - V_{c0}) \quad (4)$$

- f_{c0} is the central frequency and K_{VCO} is the VCO sensitivity.
  The instantaneous phase θ_{VCO} is defined as the integral over time of the angular frequency ω_{VCO}. Therefore, we deduce that the VCO voltage can be expressed as follow:

$$v_{VCO}(t) = A_{VCO}\ sin(\theta_{VCO}) \quad (5)$$

- The PLL is modeled by combining these three models. Three testbenches were developed to simulate the PLL behaviors with different paramerts. The following parameters were fixed in three cases:
  PHC: $K = 3$
  LPF: $H_0 = 1, f_p = 112KHz$
  VCO: $A_{VCO} = 1V, f_{c0} = 7MHz, V_{c0} = 0, K_{VCO} = 30KHz/V$
  $v_{ref}(t)$ is a sinusoidal waveform with an amplitude $A_{ref} = 1V$.

- *<u>CASE 1:</u> No Stimulation for the PLL*

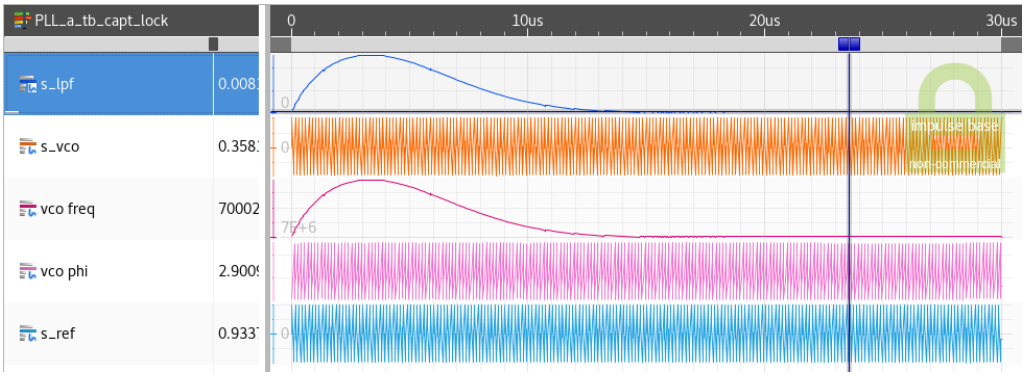

Figure 2: PLL simulation in case 1

The input voltage $v_{ref}(t)$ (s_ref) has the same frequency as the center frequency of the VCO. This means that the output signal of the VCO $v_{VCO}(t)$ (s_vco) has the same frequency as $v_{ref}(t)$and that the input voltage of the VCO $v_{ctrl}(t)$ must be equal to the parameter V_{VCO} = 0V. In reality, the PLL is a second order system, so the voltage $v_{ctrl}(t)$ (s_lpf) first goes through a transient state before stabilizing around V_{VCO}. The beginning of the stable phase defines the lock-in time which is approximatively equal to 14-15 us. When $v_{ref}(t)$ and $v_{VCO}(t)$ are synchronized at the same frequency, they have a phase shift of π/2.

- *<u>CASE 2</u>: We stimulate the PLL with $v_{src}(t)$ that takes the following constant values +1V at t = 0, -1V at t = 40µs and 0.5V at t = 80µs.*

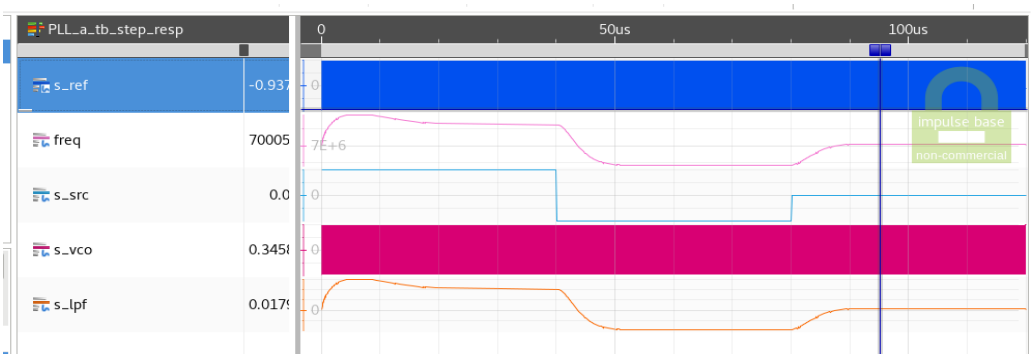

Figure 3: PLL simulation in case 2

An external VCO is used in the test model to generate a sinusoidal $v_{ref}(t)$ with three different frequencies every 40 μs. It can be seen that the control voltage of the VCO needs some time to resynchronize the $v_{ref}(t)$ and $v_{VCO}(t)$ voltages at the same frequency.

- *CASE 3*: *We stimulate the PLL with $v_{src}(t)$ that follows these characteristics: -4V <$v_{src}(t)$ <+4V by steps of 0.5V, each step lasts 35µs.*

Figure 4: PLL simulation in case 3

An external VCO generates a sinusoidal $v_{ref}(t)$ voltage with a range of frequencies (each step of $v_{src}(t)$ generates a frequency). We can see that for frequencies outside the frequency range 6.95 MHz (7 MHz - 1.5V*30KHz/V) and 7.45 MHz (7 MHz + 1.5V*30KHz/V) the PLL is unable to synchronize the $v_{ref}(t)$ and $v_{VCO}(t)$voltages. So, the tracking range is [6.95 MHz, 7.45 MHz]. A finer $v_{src}(t)$ voltage step should be used to define this range more precisely.

⇨ The SystemC AMS allows the modeling of analog systems functionality and provides good time simulations compatible with the theory results.

3 TRIAXIS ANALOG BACKEND

3.1 General Description

The aim of this subsection is to provide sufficient information about the Analog Backend components of Triaxis Gen III (MLX90421) and later versions for the purpose of assessing the SystemC AMS modeling capabilities and advantages.

Triaxis Gen III is a third generation rotatory and position sensor IC. It is a HALL effect sensor that detects the three spatial components of an applied magnetic field from which one can conclude the rotary and the translation movements [2]. In this project, I will be more interested in the Triaxis Analog Backend block which refers to the physical layer that interfaces the sensor to the wire harness. The physical layer is designed such as the same hardware can be configured to be used as an Analog Interface, a Pulse Width Modulation driver (PWM) and Single Edge Nibble Transmission driver (SENT).

The Triaxis Analog Backend of the MLX90421 is made up of these different components whose operations will be detailed in the next sub-section:

- 12-bit DAC (Digital to Analog Converter) working either in RATIOMETRIC mode (output proportional to Vext) or SENT mode (output independent of Vext) (see Figure 5)
- 1_4 multiplexer (MUX)
- RC Low pass filter
- Output driver

In Figure 5, the main components of the Backend are presented schematically, with the DAC working either in RATIOMETRIC mode or SENT mode. The multiplexer is mainly used to route on the output several test signals to increase observability during testing. The low pass filter provides relatively good filtering for the switching of the DAC and the output OPA drives the output pin directly and is configurable in several modes.

In the following subsection, I will be focusing on the DAC operation as it is the component that merges the digital and analog signals which is interesting for showing the SystemC capabilities.

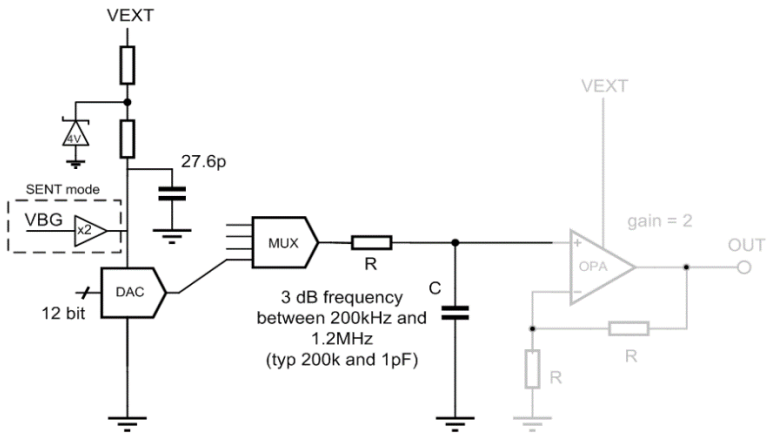

Figure 5: Triaxis analog Backend electronic circuit

3.2 Backend Components

As Already mentioned, the Triaxis Analog Backend is made of 12-bit DAC, 1-4 MUX, RC low pass filter and an output driver. In the next paragraph, I will focus on the DAC working operation as it is the only component considered in the modeling and testing.

DAC is the abbreviation of Digital Analog Converter. As its name says, a DAC is an electronic component that converts an input digital data in the form of a binary sequence, into an output analog signal, a voltage or a current in most of cases. It can be used in several fields among which we can list the audio and video systems, communication systems and mechanical devices. Usually, a DAC is described by its resolution which corresponds to the number of bits of its digital input. Its operation is also controlled by an input reference voltage which may be an external reference usually referred to the alimentation source Vext ($V_{REF.}$ in the Figure 6). In

this case, the DAC is operating in the ‘’RATIOMETRIC” mode and it follows the variations seen by the Vext voltage. In the other case, this input reference is internal to the DAC and fixed by regulator amplifiers. Therefore, the DAC is independent of the alimentation and is operating in the ‘’SENT’’ mode.

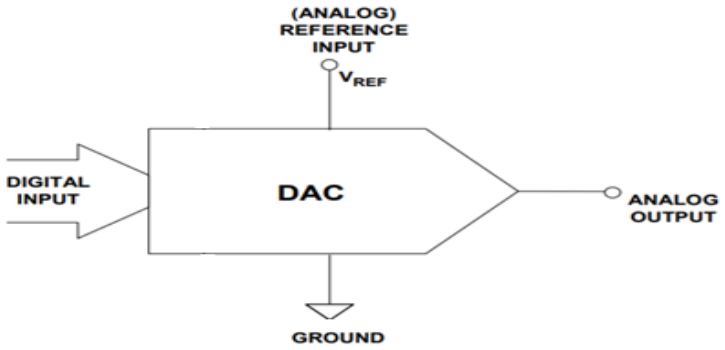

Figure 6: A simplified DAC schematic [6]

A DAC architecture can be either capacitive or resistive. In this project, a DAC based on a resistive network is considered. The circuit is a set of 12 resistors connected in series and 12 switches connecting the output node and the nodes between each two successive resistors. One can imagine the switches as CMOS transistors working either in the passing state or the blocking state. The digital control signal is obviously a 12 bits binary signal. Each bit determines the behavior of one of the switches and could visibly takes the value of “0” or ‘’1”. When a particular bit is set on the ‘’1” value, the respective switch is on and an additional current flow to the output [6] [7]. A supplementary voltage, depending on the one hand on the reference voltage and on the other hand on the position of the bit in the digital signal sequence is added to the output voltage. Therefore, the resulting output can be seen as a superposition of different voltages proportional to each other. The Figure 7 shows a simplified DAC electronic circuit.

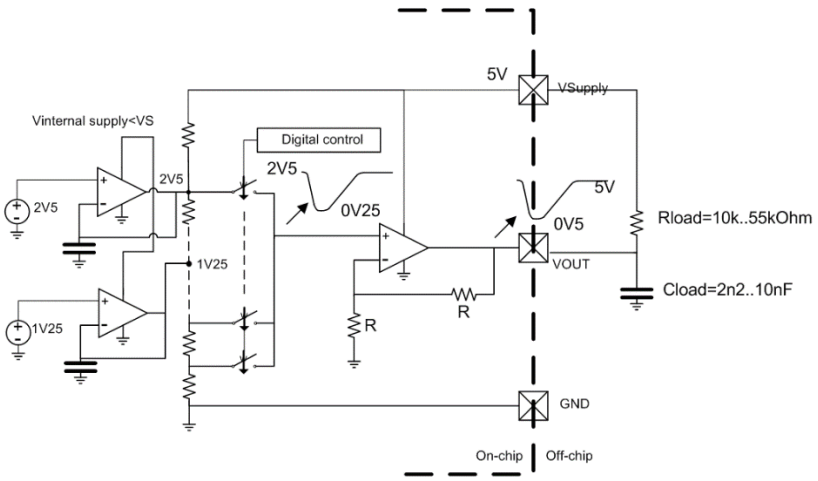

Figure 7: A simplified DAC electronic circuit

Each node has a voltage value that depends on the corresponding bit. The theoretical voltage at node i is equal to:

$$\mathbf{V_i} = \frac{\mathbf{V_{Reference}}}{\mathbf{2^{Number\ of\ bits}}}\ \mathbf{bit(i).2^i} \qquad (6)$$

Considering a digital control signal varying from all bits "0"to all bits "1" and passing by all the possible digital input combinations successively. The expected output characteristic is a staircase line seemingly a linear function if the time between the steps is negligible.

4 DIGITAL TO ANALOG CONVERTER

4.1 Ideal DAC Modeling

This model is designed to elaborate and simulate the ideal behavior of a DAC without taking into consideration any non-idealities. It is also designed to differentiate between the different modes of operation of the DAC.

The model includes 3 input TDF ports (the entire model is available in annex 6):

- DigCont: the digital control signal of the DAC, a sequence of 12 bits.
- ABE_DAC_SENT_MODE: the ABE_DAC_SENT_MODE signal which is a Boolean signal that will determine either the DAC operates in the SENT mode or in the RATIOMETRIC mode.
- Vs: The supply voltage signal.

And 1 output:

- Out: the output DAC voltage (the node of the amplifier input).

The model is developed in such a way it has a generic parameter <NBITS> which is the number of bits of the Digital Control signal. It has 1 input parameter Vsint1 which is the voltage fixed by the first regulator amplifier of the DAC. The model is based on two different functions:

- Void initialize (): the function that contains the debugging lines.
- Void processing (): the function that describes the detailed DAC operation.

In Figure 8, we can see the assignments of the processing function that provide the output value:

```
void processing(){

    double sum = 0.0 ; // sum of 2^i where i is the index of bits having the value 1
        for (int i = 0 ; i < NBITS; i++) {
            if (DigCont.read().get_bit(i) == 1 ) sum += pow(2.0,i);

        }
        if (ABE_DAC_SENT_MODE == true)
            out.write(sum*(Vsint1_/pow(2.0,NBITS))); // sent mode
        else
            out.write(sum*(Vs.read()/pow(2.0,NBITS+1))); // ratiometric mode
}
```

Figure 8: The Ideal DAC processing function (extracted from annex 6)

Depending on the value taken by the ABE_DAC_SENT_MODE signal, the DAC can operate either in the SENT mode and therefore the output is independent of the supply voltage signal (if ABE_DAC_SENT_MODE = true) and calculated according to the regulator amplifier voltages, or in the RATIOMETRIC MODE (If ABE_DAC_SENT_MODE = false) and in this second case the output is proportional to the supply voltage and adopts the same shape. For every possible input combination of the Digital Control signal, the output is calculated according to the following expression:

- SENT mode: References values are fixed by regulator amplifiers on which depends the output voltage. If we suppose that these values are stable and don't undergo any fluctuation, the following expression can be extracted (we consider only the first regulator amplifier):

$$\mathbf{V_{out}} = \frac{\mathbf{V_{sint1}}}{\mathbf{2^N}} \sum_{\mathbf{1}}^{\mathbf{N}} \mathbf{bit(i).\,2^i} \qquad (7)$$

Where V_{sint1} is the value fixed by the first regulator amplifier and N is the number of bits (NBITS).

- RATIOMETRIC mode: The regulator amplifiers are not used in this mode. The output voltage is a function of the Digital Control signal and the supply voltage Vs. In fact, the reference voltage in this case becomes $\frac{V_s}{2}$. The expression of the output is therefore equal to:

$$\mathbf{V_{out}} = \frac{\mathbf{V_s}}{\mathbf{2^{N+1}}} \sum_{\mathbf{1}}^{\mathbf{N}} \mathbf{bit(i).\,2^i} \qquad (8)$$

In the processing function, we calculate first the sum of the active bits' weights of the Digital Control signal and then we multiply it by the analog output value corresponding to the least significant bit (for more details about the concept of bit weights check section 4.3.2).

Two testbenches were developed in order to visualize the DAC operation in the SENT mode, as well as in the RATIOMETRIC mode (see annexes 7 and 8). The system and source parameters were set up in the same way in both testbenches:

- The supply voltage is a sinusoidal source (see annexes 1 and 2 for more details). The amplitude is fixed to 5V and the frequency to 1MHz.
- The regulator amplifier voltage Vsin1 is fixed to 2.5V.
- ABE_DAC_SENT_MODE is fixed either to true for the SENT mode or to false for the RATIOMETRIC mode.
- The DigCont is a ramp function (linear interpolation) that permits the generation of all possible input combinations, which means all the unsigned 12-bit sequences between 0 and $2^{12} - 1$. This operation is possible thanks to a SC_MODULE that was developed independently (see annex 3 for more details).
- The Gain of the amplifier is equal to 2 (see annex 4 for more details).

The simulation parameters are detailed as follows:

- The number of points per period is equal to 50. This parameter signifies the number of times the output value will be computed during the simulation in one period. In another word, the number of samples in one period. Each increase in this parameter will induce an amelioration of the resolution.
- The TSTEP is equal to 1/Frequency/number of points per period. This parameter is the time taken between two successive evaluations of the output (between two samples). The process is launched every TSTEP.
- The simulation frequency is equal to the supply voltage frequency (sinusoidal source).
- The CLK_PER parameter is the clock period and equal to 1/Frequency/number of points per period.
- TSTOP is the time of the end of simulation and is therefore equal to CLK_PER*2^{12}.

Figure 9 and Figure 10 show the simulations results. These simulations are done in the Eclipse C++ environment and the waveforms are obtained from VCD files with the Eclipse Impulse plug-in:

- SENT mode:

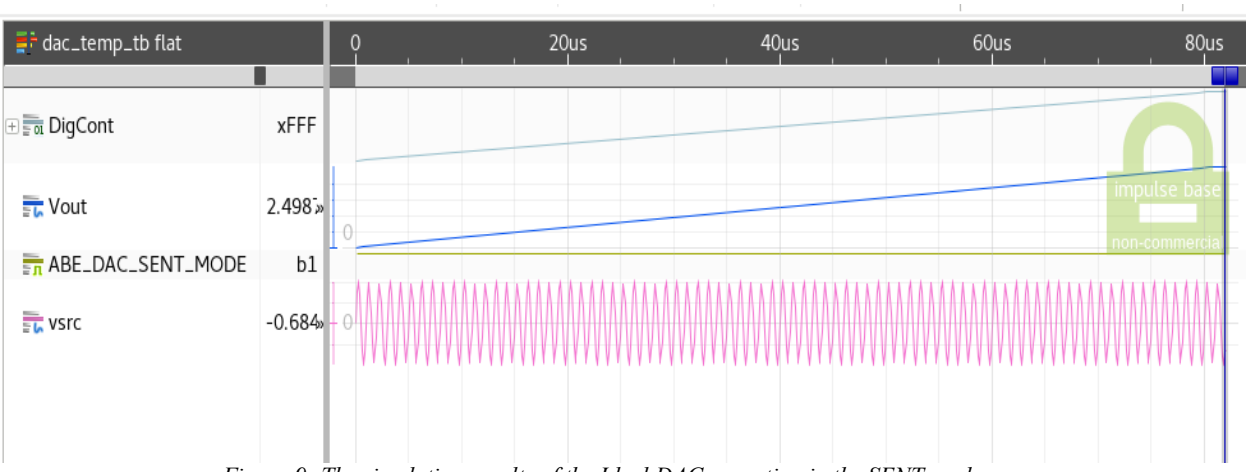

Figure 9: The simulation results of the Ideal DAC operation in the SENT mode

For all the simulation waveforms, Vsrc is a sinusoidal source voltage and which is mapped to Vs. I used a sinusoidal source so that the dependency of the output signal on the supply voltage in the RATIOMETRIC mode can be observed.

The output adopts the expected shape. When the Digital Control signal is a ramp, it means that the 12-bit sequence increase by 1 at each TSTEP, the output appears as a linear function (I used a linear interpolation mode) which ranges between a minimum equal to 0 that corresponds to the binary code of 0 and a maximum equal to 2.498V which corresponds to the binary code of $2^{12} - 1$ (the theoretical value is equal to 2.5V).

- RATIOMETRIC mode:

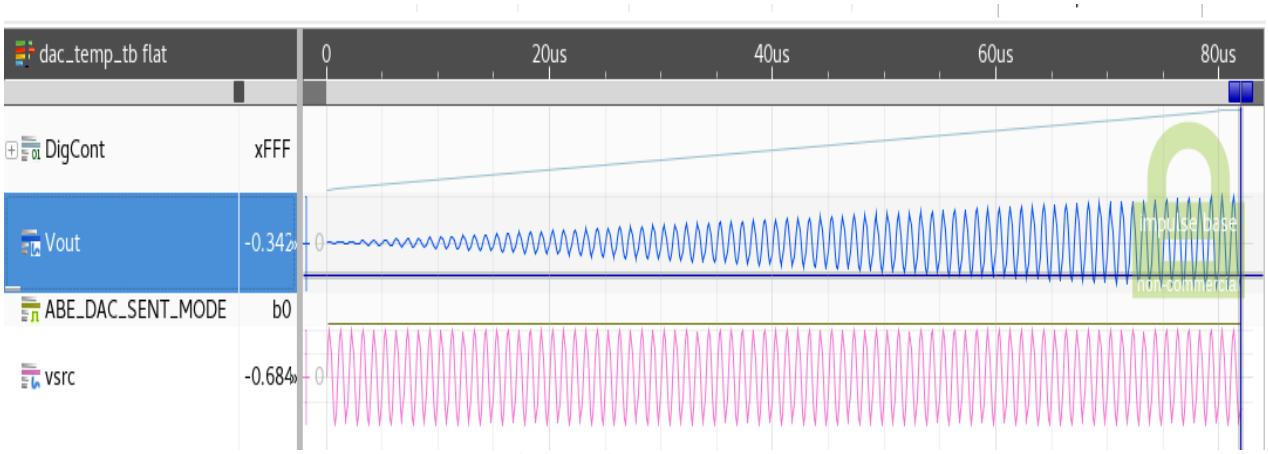

Figure 10: The simulation results of the Ideal DAC operation in the RATIOMETRIC mode

For the RATIOMETRIC mode, the output is no more a linear function. However, it is a function of the Digital Control signal and Vs signal simultaneously. In this case also, the output varies between two extremums, a minimum equal to 0 and a maximum equal to 2.5V (this value corresponds to the sinusoidal source amplitude divided by 2).

⇨ These results show that our model meets the requirements cited by the theory. It emphasis also the capabilities of SystemC AMS for implementing different operation modes of the same component in the same model without going into the details of the electronic components that form the DAC.

4.2 DAC temperature dependency modeling

The previous model describes the DAC behavior in the ideal case. One of the non-idealities I have considered is the dependency of the output signal on the temperature. A model of the DAC depending on the temperature variation is developed and tested in this sub-section.

Studying the temperature fluctuation can be explained by several factors. First of all, the DAC analog circuit consists of a network of resistors connected in series. The resistance is sensitive to temperature. For most materials the resistivity increases with temperature. An exception is semiconductors in which the resistivity decreases with temperature [8]. Secondly,

the temperature can even affect the flow of electrons through the conducting wires and subsequently the electrical current at the output. Therefore, it comes the idea to design a model that will consider the temperature fluctuation over time. The difficulty of this task is the development of a model which studies the temperature variation and its influence on the output characteristic without adding a port which communicates the value of the temperature simultaneously at the time to the rest of the system. In fact, the DAC architecture of the HALL effect sensor provided by Melexis does not consider any port for the temperature. Then no external access to control this parameter exists. Afterwards, it was necessary to look for a mean to vary the temperature in the testbench and in a way independent of the generic model developed. Within this context, several tracks with different methodologies have been proposed and studied. Some have led to results and others have not.

In this section, I will discuss the approach that leads to the desired results in details. It consists mainly of a DAC model that considers the temperature variation and a testbench that allows to verify that the temperature variation is properly modeled. The model consists of:

- The input and output signals are the same used in the Ideal DAC model.
- The same expression of the output as a function of the Digital Control signal is used.
- We add the T_amb parameter initialized in the module constructor and defined as a private variable which defines the ambient temperature value.
- Two functions are implemented, the void initialize () function and the void processing () function.
- The temperature value is communicated to the module thanks to a set.Temp () function.

For the sake of simplicity, a linear relationship between temperature and output has been considered as it can be observed in the processing function in Figure 11:

```
}
 void processing(){

        double inter;
        double sum = 0.0 ;
            for (int i = 0 ; i < NBITS; i++) {
                if (DigCont.read().get_bit(i) == 1 ) sum += pow(2.0,i);

            }
            if (ABE_DAC_SENT_MODE == true)
                inter = (sum*(Vsint1_/pow(2.0,NBITS)));
            else
                inter = (sum*(Vs.read()/pow(2.0,NBITS+1)));

            out.write (inter-inter*(temp - T_amb_)/ T_amb_);

            using namespace std;
            cout << "-- " << temp <<" temp " << endl;

    }
```

Figure 11: The DAC processing function considering the temperature variation (extracted from annex 9)

If the external temperature is equal to the ambient temperature then the output is equal to the ideal value. And if the external temperature is different from the ambient temperature, the output value is multiplied by the factor:

$$\textit{(temperature - ambient temperature)/ ambient temperature)} \qquad (9)$$

The goal of this approach is to implement a solution that allows us to dynamically communicate the temperature to the DAC model while the signals mapping is taking place. The idea is to develop a module that contains all the testbench instructions (among them signals mapping) as well as the function updating the temperature values. In this module we define a SC_THREAD PROCESS (see Figure 12) that allows the temperature variation simultaneously.

```
template <int NBITS> // NBITS digital control bits

class tempvariation: public sc_module {
public:

    sc_in<bool> clk;
    sc_in<bool> clk_temp;
    sca_tdf::sca_signal<double> Vout;
    sca_tdf::sca_out<double> Voutampl;
    sca_tdf::sca_signal<double> vsrc;
    sc_signal<bool> ABE_DAC_SENT_MODE;;
    sc_signal<sc_bv<NBITS> > DigCont;

    Sinesrc Sin;
    Dac_temp<NBITS> Dac;
    inputcombination<NBITS> digital;
    Amplification amp;


     void temperature_update () {

            double T;
            double x;
            while (true)
             {
            x =  ((double)std::rand())/RAND_MAX ;
            T = T_MIN_ + x * (T_MAX_-T_MIN_);
            Dac.set_Temp(T);
            wait();
            using namespace std;
            cout << "-- " << T << " T " << endl;
             }
       }
    SC_HAS_PROCESS(tempvariation);
```

Page 1

```
tempvariation(const sc_core::sc_module_name& name,
            double VSINT1 = 2.5 , // [V]
            double GAIN = 2.0,
            double T_AMB = 37.0,
            double AMPL = 1.0,
            double OFFS = 0.0,
            double FREQ = 1.0e6,
            double T_MAX = 40.0,
            double T_MIN = 35.0)

  :  VSINT1_(VSINT1), GAIN_(GAIN), T_AMB_(T_AMB),AMPL_(AMPL),
     OFFS_(OFFS),FREQ_(FREQ),T_MAX_(T_MAX),T_MIN_(T_MIN),
     clk_("clk"), Vout("Vout"), Voutampl("Voutampl"),vsrc("vsrc"),
     DigCont("DigCont"),ABE_DAC_SENT_MODE("ABE_DAC_SENT_MODE"),
     Sin("Sin",AMPL_, FREQ_, OFFS_),Dac("Dac",VSINT1_,T_AMB_),digital("digital"),
     amp("amp")

{
    ABE_DAC_SENT_MODE.write(true);
// sin component
        Sin.out(vsrc);
// Digital control component
        digital.clk(clk);
        digital.output(DigCont);
// DAC component
        Dac.Vs(vsrc);
        Dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
        Dac.DigCont(DigCont);
        Dac.out(Vout);
// amplification component
        amp.in(Vout);
        amp.out(Voutampl);

        SC_THREAD(temperature_update);
        sensitive << clk.pos();
```

Page 2

```
    virtual ~tempvariation() {}

protected:

private:
        double VSINT1_ ; // [V]
        double GAIN_ ;
        double T_AMB_ ;
        double AMPL_ ;
        double OFFS_ ;
        double FREQ_ ;
        double T_MAX_ ;
        double T_MIN_ ;
};
```

Page 3

Figure 12: The tempvariation module (extracted from annex 10)

In the tempvariation module, we define the input and output signals of the DAC module, the input and output signals of the other modules (sinusoidal source module, Amplifier module, Digital sequence generation module) but also the testbench signals (clk).

We define the "temperature_update" as the function that enables the generation of a random value of the temperature between two extreme values. All the parameters that used to be fixed inside the testbench as in the case of the ideal DAC testbench are declared this time as private variables and initialized in the module constructor. The signals mapping is performed inside the tempvariation module. We can observe the obtained results for both the SENT mode and the RATIOMETRIC mode in the Figure 13, Figure 14 and Figure 15:

- SENT mode:

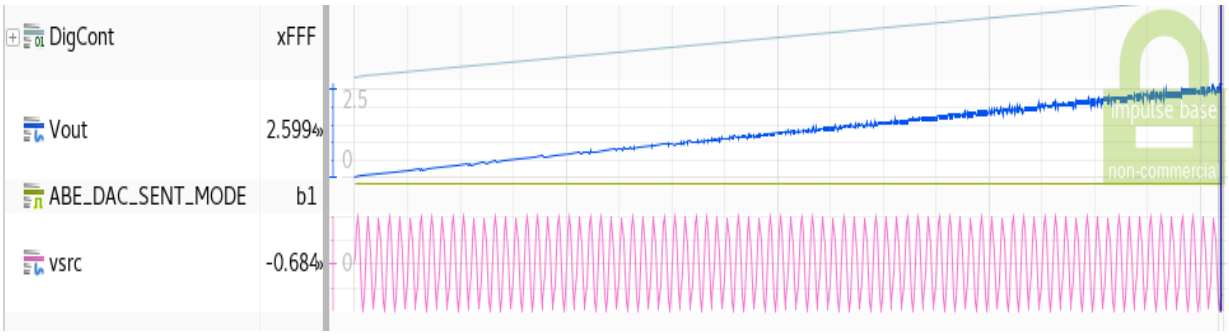

Figure 13: The simulation results of the DAC operation in the SENT mode considering the temperature variation

- RATIOMETRIC mode:

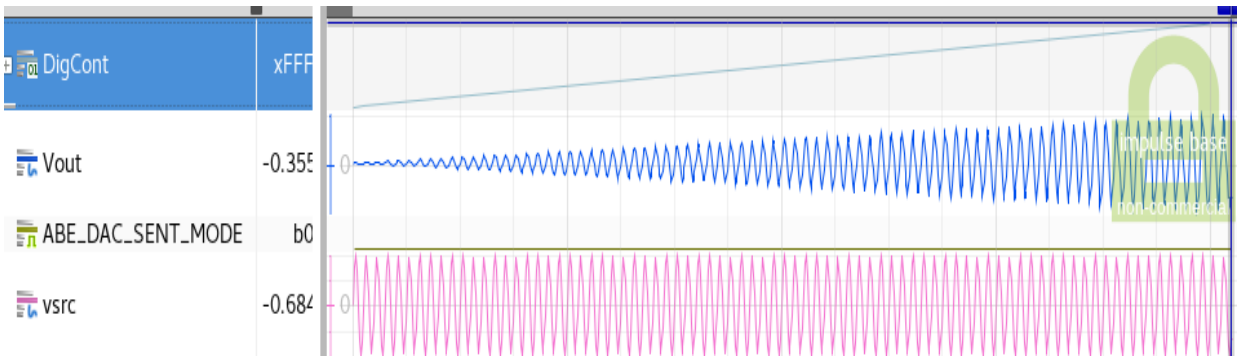

Figure 14: The simulation results of the DAC operation in the RATIOMETRIC mode considering the temperature variation

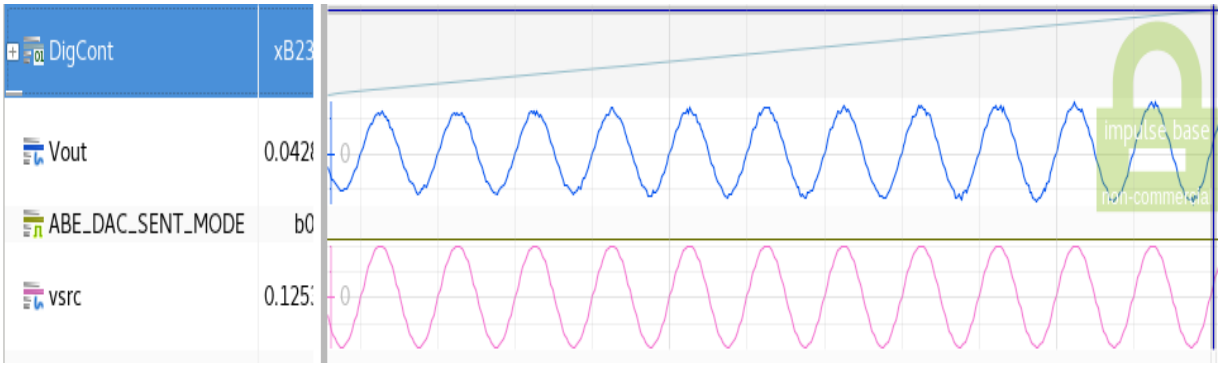

Figure 15: ZOOM on figure 14

We notice that the output no longer follows a straight line (SENT mode) or a sinusoidal shape (RATIOMETRIC mode) but rather the values taken by the characteristic fluctuate around the ideal characteristics. It's as if a noise is detected at the output and which according to our model is explained by the temperature variation that impacts the DAC output signal.

⇨ The SystemC AMS allows the modélisation of an external parameter impact on the DAC operation. Without adding a port, one can vary the temperature simultaneously with the simulation at each timepoint.

4.3 INL and DNL modeling

In the previous sub-section, non-ideal DAC considering the temperature dependency was modelized. In this sub-section, another non-ideality will be considered which is the INL and the DNL effect on the DAC behavior.

4.3.1 About INL/DNL

The INL (integral nonlinearity) and the DNL (differential nonlinearity) are specifications that describe static DAC non-idealities. They are measured to qualify and to test the DAC performance [5] [11]. In fact, when the DNL and INL have relatively high values, one can foresee a characteristic at the output which deviates with a certain error from the ideal characteristic. For a better explanation of the DNL and the INL, we introduce the concept of the bit weight. For this purpose, considering the DAC studied in this project. Each bit of the Digital Control signal has a particular weight that reflects its contribution to the output signal. For a 12 bits sequence $b_{11}\ b_{10}\ldots.b_i\ldots.b_1 b_0$, b_0 is called LSB (the least significant bit) and b_{11} is called MSB (the most significant bit). Within this context, LSB can refer to the analog value of the least significant bit. The weight of the bit indexed i is equal to the value given by the following expression: 2^i* LSB. These values are deduced from the analog circuit formed by the resistances network connected in series. Consequently, a simple disturbance causing a small variation in the value of one of the resistances has a direct impact on the weight of the corresponding bit and therefore on the output signal. The DNL and INL are measurements which reflect in an observable and interpretable way a physical phenomenon related to the fluctuations of the values of the resistances which in turn can be due to many factors, notably the temperature. In an ideal context, when increasing the Digital Control signal by one, the output should increase by one unit (the analog value corresponding to one LSB). In a more simplified manner, the DNL is the maximum deviation of the output steps from the ideal analog LSB value [5] [11].

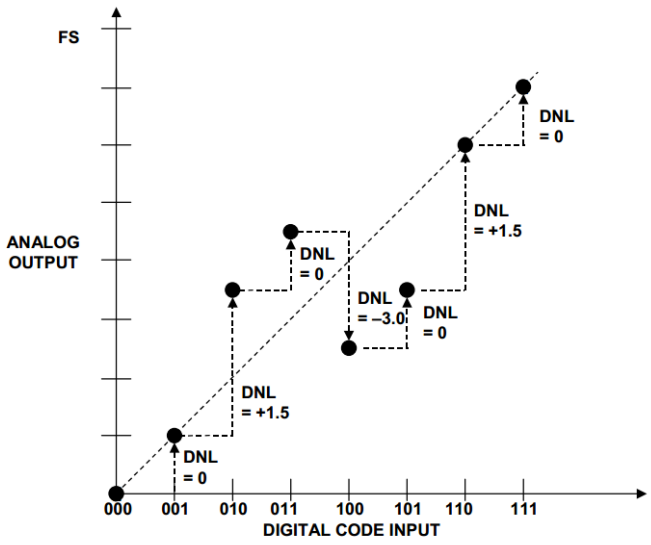

Figure 16: Figure showing DNL measurement for a 3-DAC [5]

Considering the two successive codes “011” and “100” in Figure 16, the DNL of such transition is equal to the difference between the analog output value of the code “100” and the previous analog output value, more precisely the one corresponding to the code “011”, to which we add one unit. Therefore, the DNL can be either positive or negative. After calculating the DNLs of each transition, the final DNL specification of the DAC can be deduced as the absolute value of the maximum DNLs already calculated. The DNL is measured in LSB in the most of cases. Yet, it can be expressed also on volt, ampere or as a percentage of the full-scale value [5] [11]. For the INL, it can be defined as the maximum deviation of the output characteristic from the ideal characteristic. In the example shown above, the ideal transfer characteristic is a ramp (due to the fact that the Digital Control signal takes successive code combinations). In this particular case, the INL is equal to 1.5 LSB. This value is achieved for the four codes: “010”, “011”, “100”and “101”.

4.3.2 INL/DNL modeling

In order to design the INL and DNL non-ideality, I opted for two different approaches. A TOP DOWN approach which is an exploration approach. Within this context, the DNL and INL parameters of the DAC are communicated to the model as inputs parameters and therefore the values taken by the output are constrained and forced by these two bounds. This first consideration results in an output characteristic respecting the specifications margins. The second one is a BOTTOM UP approach which is a verification approach. This time, a small random variation around each bit weight is fixed and the DNL and INL are calculated and extracted from the output signal.

- TOP DOWN Approach:

The DNL and the INL of the 12-bit DAC are fixed to 1.5 LSB and 2 LSB. In order to model this concept, I started with an approach that allows to visualize a characteristic that respects both input conditions but is only valid for a Digital Control signal whose function over time is a ramp (the entire module is available in annex 13). That means that this model is not generic and which needs improvements to be adapted to any possible digital input combination sequences.

This model is based on a while loop. In fact, A random generation of a variation between -INL and INL is added to the ideal output value using the srand () function. This operation is repeated until the second condition controlled by the DNL is verified. As the DNL condition is related to both the previous and the current output value, I created a traceable variable in order to store the previous calculated value of the output and compare it to the current computation at each iteration of the while loop. This traceable variable can be plotted and visualized in the simulation waveforms.

```
void processing(){

    double x;
    double y;
    double sum = 0.0;
    double output_value = 0.0;
    for (int i = 0 ; i < NBITS; i++) {
        if (DigCont.read().get_bit(i) == 1 ) {
            sum += pow(2.0,i) ;}
    }
    do {
        x =  ((double)std::rand())/RAND_MAX ; // double between 0 and 1
        y = -INL_ + x * (INL_ - (-INL_));
        output_value = sum + y;
        current_dnl = previous_output - output_value + 1 ; }

     while (abs (current_dnl)> DNL_);


    if ( abs (y) >INL_var_) INL_var_ = abs (y);
    if (abs (current_dnl) >DNL_var_) DNL_var_ = abs(current_dnl);

    using namespace std;

    cout << "-- " << current_dnl << " current_dnl " << endl;
    cout << "-- " << y << " variation around ideal value " << endl;
    cout << "-- " << DNL_var_ << " maximum value of DNL till now " << endl;
    cout << "-- " << INL_var_ << " maximum value of INL till now " << endl;


    if (ABE_DAC_SENT_MODE == true)
        out.write((output_value)*(Vsint1_/pow(2.0,NBITS)));
    else
        out.write((output_value)*(Vs.read()/pow(2.0,NBITS+1)));

    previous_output = output_value;

}
```

Figure 17: The DAC processing function considering the INL/DNL variation: TOP DOWN approach (extracted from annex 13)

As it can be shown in the model above, x is a local double variable of the processing function which takes at each timepoint a random value between 0 and 1. y which is a local double variable as well, takes the value of the random variation between -INL and INL and it is added to the ideal sum of the active bits' weights (we remind that the analog DAC output ideal value is equal to this sum multiplied by the analog value corresponding to 1 LSB). This process is repeated in a while loop until the DNL condition is respected. At the end of the function, the current output value is stored in the previous_output variable. The DNL and INL variations values can be observed thanks to the two private variables DNL_var_ and INL_var_.

- As for the Ideal DAC model, three input TDF signals and one output TDF signal are defined.
- A traceable variable that we call current_dnl is added to the model in order to visualize the DNL variation during the simulation.
- We add to this model two input parameters and private variables other than Vsin1, which are the DNL and the INL.

A testbench is developed to observe the output characteristic as well as the DNL variation over the time (see annex 14 for more details). For the sake of simplicity, the model will be simulated in the SENT mode (the simulation was also done for RATIOMERIC mode and results respecting the DNL and INL margins were observed). The same specifications as the first testbenches are used. In addition, the DNL is set to 1.5 LSB and the INL is set to 2 LSB.

The Figure 18 and Figure 19 show the simulation results:

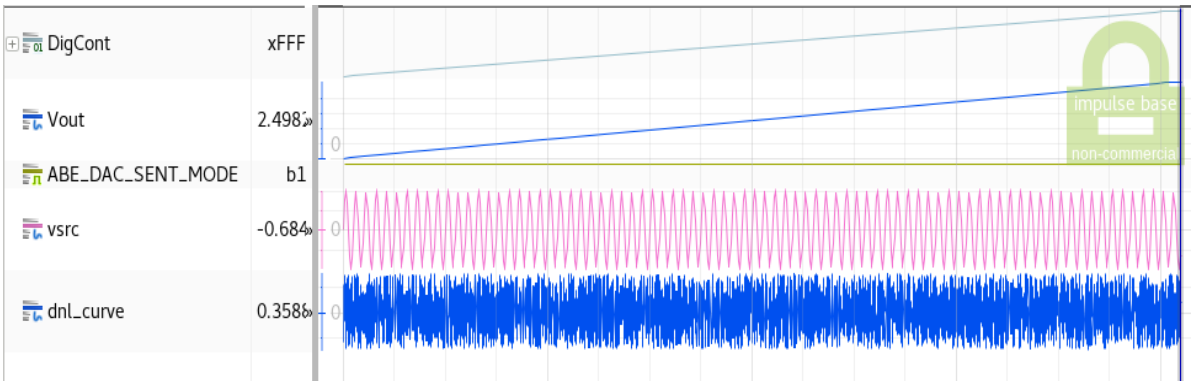

Figure 18: The simulation results of the DAC operation in the SENT mode considering the DNL/INL (TOP DOWN approach)

In general, it can be seen from the Figure 18 that the output follows a straight line as long as the Digital Control signal is a ramp. However, the dnl_curve varies between a maximum equal to 1.49976 and a minimum value -1.49976 in a random way and this confirms the designed model.

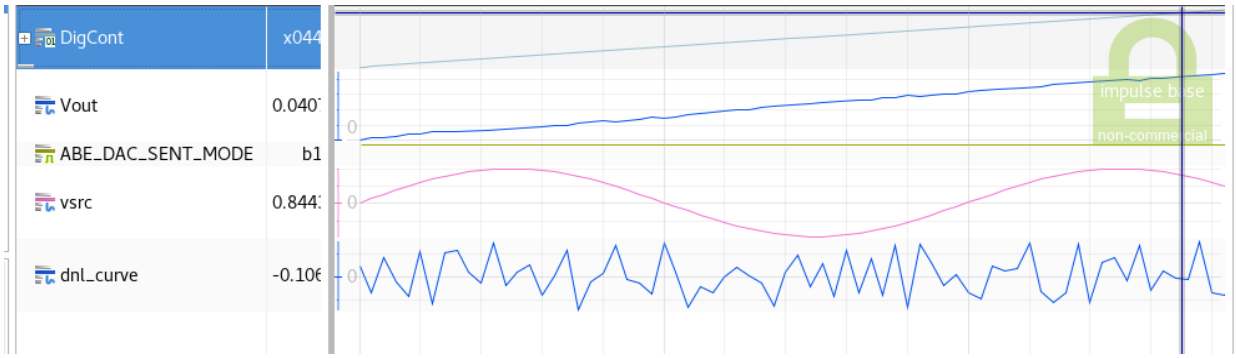

Figure 19: ZOOM on Figure18

By zooming in on the output characteristic, it can be seen that the characteristic no longer follows a straight line but rather a curve that varies around a linear line. These small variations are errors due to considerations made regarding INL and DNL.

```
-- 1.22826 current_dnl
-- -0.548153 variation around ideal value
-- 1.49976 maximum value of DNL till now
-- 1.9978 maximum value of INL till now
-- 0.358846 current_dnl
-- -0.906998 variation around ideal value
-- 1.49976 maximum value of DNL till now
-- 1.9978 maximum value of INL till now
-- -0.891059 current_dnl
-- -0.0159388 variation around ideal value
-- 1.49976 maximum value of DNL till now
-- 1.9978 maximum value of INL till now
```

Figure 20: an excerpt of the values taken by the current_dnl, the DNL and INL of the DAC

As for the INL and DNL, Figure 20 shows that they fluctuate in a random way which affirms the model developed. The maximum value taken by the absolute value of the variation of DNL is equal to 1.49976 LSB and that of INL is equal to 1.9978 LSB. These two values meet the given specifications (1.5 for INL and 2 for DNL).

This simplified model allows us to observe the effect of DNL and INL on the output of the DAC but which is unfortunately only valid for a particular digital input sequences order.

In order to make improvements on this model, another approach is considered and which provides credible results for any input combinations of the Digital Control signal (the entire model is developed in Annex 17).

For this purpose, two arrays of 2^{NBITS} elements are created. The first array contains the ideal values of the sum of the active bits' weights (we remind that the analog DAC output ideal value is equal to this sum multiplied by the analog value of corresponding to 1 LSB) for all possible codes of the Digital Control signal, while the second array contains the already calculated ideal values to which we add a small perturbation that considers the INL and DNL. This is also done for all possible codes of the Digital Control signal. This process is initialized in the initialize function (see Figure 21) which is called only once at the end of the elaboration and before starting the time simulation. The while function this time is implemented inside the initialize function and not in the processing function and it is repeated until the values of the second arrays respect the INL and DNL specifications.

```
void initialize() {

    // function to guarantee the generation of random values
    srand(999);
    for (int i = 0 ; i < (int)(pow(2.0,NBITS)) ; i++) {

        double var_rand; // random value between 0 and 1
        double var;     //  random value between DNL and -DNL
        double difference = 0.0; // difference between ideal value and not ideal value to conclude the INL
        sc_bv<NBITS> x;
        x = (sc_uint<NBITS>)(i); // coder l'entier i sur NBITS

        // Reference table of ideal values

        double sum_ideal = 0.0;
        for (int j = 0 ; j < NBITS; j++) {
            if (x[j] == 1 ) {
                    sum_ideal += pow(2.0,j) ;}
            }
        tab_ref_ideal[i]=sum_ideal;

        // Reference table of not ideal values respecting DNL and INL

        if (i==0) { tab_ref_not_ideal[i]=0 ;}
        else {
            do {
                    var_rand = ((double) rand()/RAND_MAX);
                    var = ( -DNL_ + var_rand * ( DNL_ + DNL_));
                    tab_ref_not_ideal[i] = tab_ref_not_ideal[i-1] + 1 + var;
                    difference = tab_ref_not_ideal[i] - tab_ref_ideal[i];
                    }
            while (abs(difference) > INL_);
                }
```

Figure 21: The DAC initialize function considering the INL/DNL variation: TOP DOWN approach improved version (extracted from annex 17)

These two arrays are then used in the processing function (see Figure 22) in order to carry out at each input the value of the corresponding non-ideal analog output, also to plot the variation of the DNL.

```
void processing(){


    if ((sc_int<NBITS> (DigCont.read()) > 0)){

         if (ABE_DAC_SENT_MODE == true)
            out.write(tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())]*(Vsint1_/pow(2.0,NBITS)));

          else
            out.write(tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())]*(Vs.read()/pow(2.0,NBITS+1)));

          current_dnl = tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())] -
                    ( tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())-1] +1);}

    else if ((sc_int<NBITS> (DigCont.read()) <0)){
         if (ABE_DAC_SENT_MODE == true)
            out.write(tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())+(int)pow(2.0,NBITS)]*(Vsint1_/pow(2.0,NBITS)));

         else
            out.write(tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())+(int)pow(2.0,NBITS)]*(Vs.read()/pow(2.0,NBITS+1)));
         current_dnl = tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())+(int)pow(2.0,NBITS)] -
                    ( tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())+(int)pow(2.0,NBITS)-1] +1);}
    else current_dnl = 0.0;
    using namespace std;
    cout << current_dnl <<endl;

}
```

Figure 22: The DAC processing function considering the INL/DNL variation: TOP DOWN approach improved version (extracted from annex 17)

A testbench is developed to observe the output characteristic as well as the DNL variation over the time (see annex 18 for more details). For the sake of simplicity, the model will be simulated in the SENT mode (the simulation results in the RATIOMETRIC mode were also correct). The same specifications as the first testbenches were fixed. And the DNL and INL are fixed to 1.5 LSB and 2 LSB. I was centered to use a ramp function for the Digital Control signal to better interpret the simulation results. We obtain the following waveforms in Figure 23 and Figure 24:

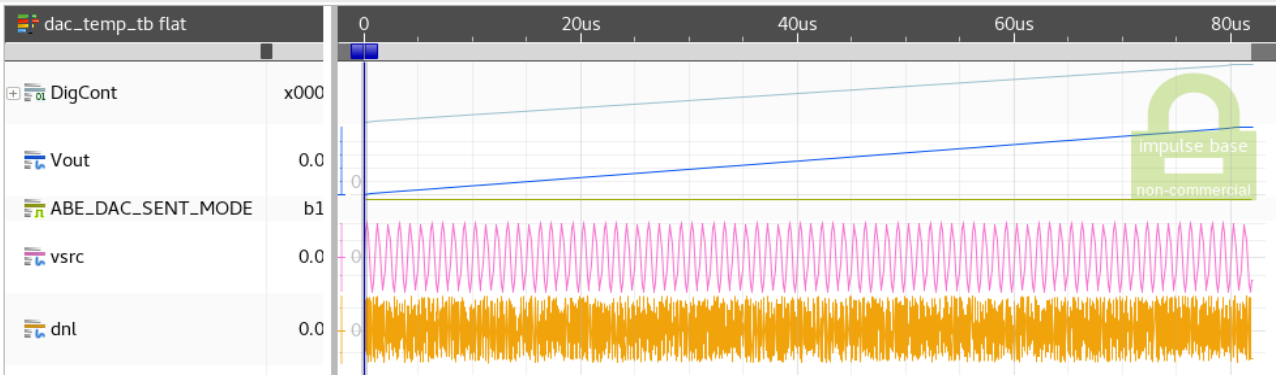

Figure 23: The simulation results of the DAC operation in the SENT mode considering the DNL/INL: TOP DOWN approach improved version

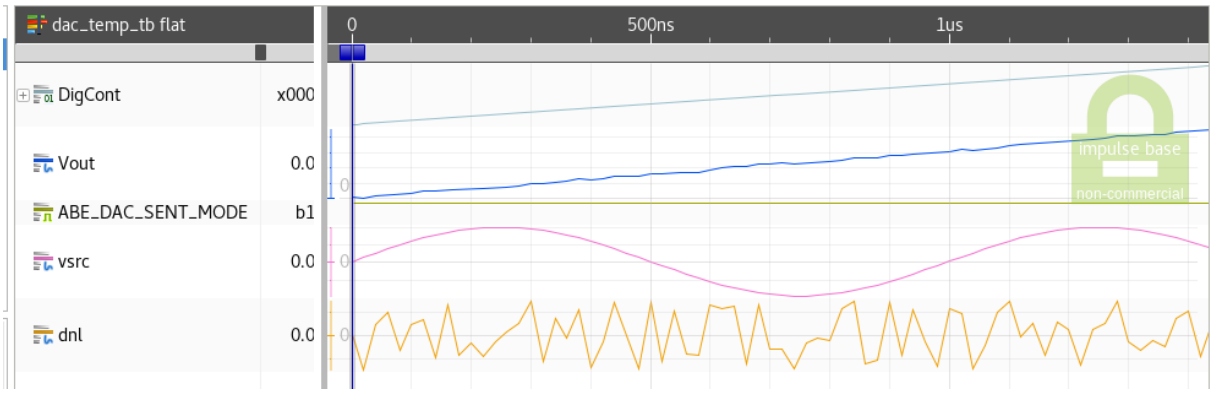

Figure 24: ZOOM on Figure 23

We get the same confirmed results as the previous model. However, this time we have developed a generic model valid for any digital input. Simulations were performed for random and sinusoidal signal inputs and correct results were also observed.

- BOTTOM UP approach:

Unlike the first approach, which was more of an exploration and confirmation method, this approach seeks to verify and extract DNL and INL values from a model by applying random variation around each bit of the Digital Control Signal. For this purpose, a model was developed in a different way (see annex 15 for more details). Indeed, two arrays of 12 elements and 2^{NBITS} elements that we called weight_bit and tab_ref_ideal has been defined as private variables and filled in the initialize function. The first contains the weight of each bit perturbed by a certain random value. This table will be the basis for calculating the output signal values in the processing function. The second array contains in contrast the ideal DAC output values (sum of the active bits weights) for all possible input combinations which will allows us to extract the INL of the DAC.

We define a private variable that we call previous_output in which we stock dynamically the previous calculated output value in order to calculate simultaneously the DNL of the DAC.

```
void initialize() {

    // function to guarantee the generation of random values
    srand(999);
    double var_rand; // random value between 0 and 1
    double var =0.0;

    for (int j = 0 ; j < NBITS; j++) {
        var_rand = ((double) rand()/RAND_MAX);
        var = -0.37 + var_rand*0.74;
        weight_bit[j] = pow(2.0,j) + var ;}

    for (int i = 0 ; i < (int)(pow(2.0,NBITS)); i++) {

        sc_bv<NBITS> x;
        x = (sc_uint<NBITS>)(i); // coder l'entier i sur NBITS

        double sum_ideal = 0.0;
        for (int j = 0 ; j < NBITS; j++) {
            if (x[j] == 1 ) {
                    sum_ideal += pow(2.0,j) ;}
            }
        tab_ref_ideal[i]=sum_ideal;}
```

Figure 25: The DAC initialize function considering the INL/DNL variation: BOTTOM UP approach (extracted form annex 15)

```
void processing(){

    double output = 0.0;
    double sum = 0.0 ;
    for (int i = 0 ; i < NBITS; i++) {
        if ((sc_uint<NBITS>)DigCont.read().get_bit(i) == 1 ) {
        sum += weight_bit[i] ;}
            }

    output = sum ;
    if (ABE_DAC_SENT_MODE == true)
        out.write(output*(Vsint1_/pow(2.0,NBITS)));

    else
        out.write(output*(Vs.read()/pow(2.0,NBITS+1)));

    if (sc_int<NBITS> (DigCont.read()) < 0){
        dnl_variation = output - (1 + previous_output);
        inl_variation = output - tab_ref_ideal[(sc_int<NBITS>) (DigCont.read()) + (int) pow(2.0,NBITS)];}

    else
         {dnl_variation = output - (1 + previous_output);
          inl_variation = output - tab_ref_ideal[(sc_int<NBITS>)(DigCont.read())];}

    if (abs (DNL_)< abs (dnl_variation)) DNL_ = dnl_variation;
    if (abs (INL_) < abs (inl_variation)) INL_ = inl_variation;


        using namespace std;
            cout << "-- " << "DNL" << DNL_ << endl
                 << "-- " << "INL" << INL_ << endl;
    previous_output=output;
`
```

Figure 26: The DAC processing function considering the INL/DNL variation: BOTTOM UP approach (extracted from annex 15)

In the processing function, the values of the output signal are calculated as follows. If the bit i=1, the value of weight_bit [i] is added to a local variable sum initialized to 0.

dnl_variation and inl_variation are two traceable variables that compute the values of the DNL variation as well as the INL variation in each sample execution. They are after compared to the maximum DNL variation and the maximum INL variation seen by the simulation in order to conclude about the final DNL and INL of the DAC.

- The dnl_variation is calculated as the difference between the current output and the previous output to which we add 1 LSB.
- The inl_variation is calculated as the difference between the ideal value of the output and the actual value computed.

A testbench with the same specification as the first testbenches was developed in the SENT mode (see annex 16 for more details) to visualize the simulation results in Figure 27 and Figure 28:

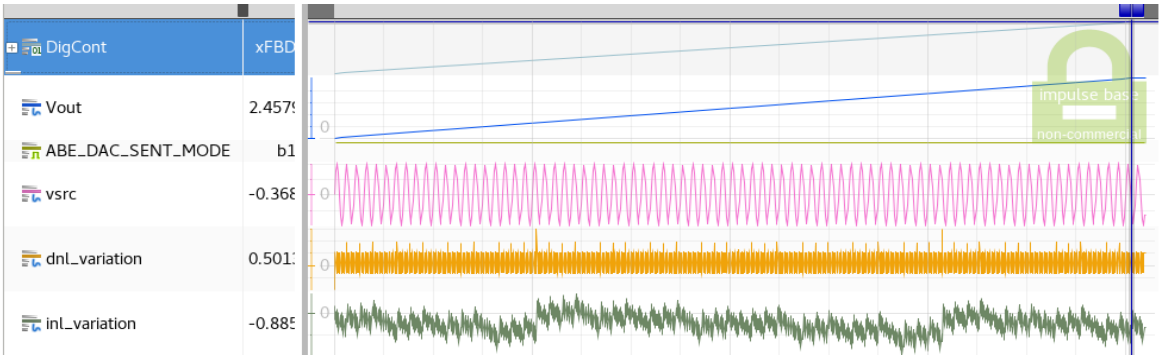

Figure 27: The simulation results of the DAC operation in the SENT mode considering the DNL/INL: BOTTOM UP approach

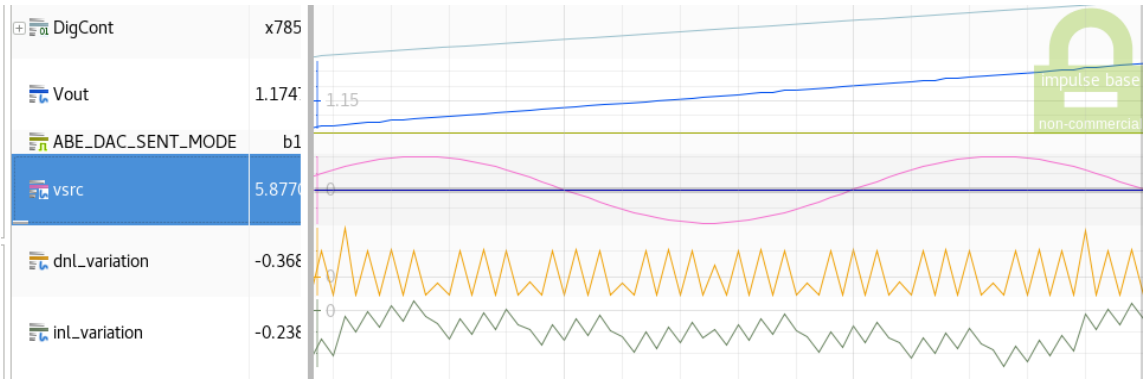

Figure 28: ZOOM on Figure 27

We obtain the same curves as the TOP DOWN approach which is a predictable result. In fact, the output varies in a random way around the ideal characteristic which is due this time to the variation of the bit weights.

```
-- INL-1.74722
-- DNL1.469
-- INL-1.74722
-- DNL1.469
-- INL-1.78725
-- DNL1.469
-- INL-1.78725
-- DNL1.469
-- INL-1.78725
-- DNL1.469
-- INL-1.78725
-- DNL1.469
-- INL-1.78725
-- DNL1.469
-- INL-1.78725
-- DNL1.469
-- INL-1.78725
-- DNL1.469
-- INL-1.78725
-- DNL1.469
-- INL-1.96068
-- DNL1.469
-- INL-1.96068
-- DNL1.469
-- INL-1.96068
-- DNL1.469
-- INL-1.96068
```

Figure 29: Extraction of the DNL and INL values

The INL of the DAC is equal to 1.96068 LSB and the DNL is equal to 1.469 LSB. These values close to the theoretical 12-DAC specifications are obtained thanks to a well-determined variation around the weight of each bits. Indeed, I changed the limits of this random variation until I obtained specifications that are quite close to the desired DNL and INL. This variation is done around 0.37 and -0.37. This result is quite interesting because from a fixed value of DNL and INL we can have an idea about the limit of variation of the bit weights of any DAC and then conclude about the physical cause of these fluctuations.

- ⇨ This part is very important in the sense that it affirms the ability of SystemC AMS to design models that follow complementary approaches (TOP DOWN and BOTTOM UP).

4.4 Nonideal DAC modeling

In this sub-section I will present two final non-ideal models that consider the different operating modes of the DAC (SENT mode and RATIOMETRIC mode), the temperature variation on one side and the influence of DNL and INL on the other side. The first model is designed following a TOP DOWN approach and the second one a BOTTOM UP approach (see annexes 19 and 23 for more details). These models are developed to be integrated later by Melexis into their working environment.

Two testbenches were developed for each model. The first one allows the visualization of the temperature dependency and the second one allows to show the INL and DNL contribution (see annexes 25, 24, 22 and 21 for more details). It is possible to develop a testbench that illustrates the impact of both non-idealities on the DAC operation.

- BOTTOM UP approach:

In the first case, DNL and INL contribution must be cancelled. To do this, we add two input parameters to the model called min_var and max_var. These two variables will limit the variation of the bit weight (it was fixed at 0.37 in the previous section). By setting min_var and max_var to 0 in the testbench, we will observe only the effect of temperature. The same method is adopted in the second testbench but this time we set T_MAX and T_MIN to 0.

- TOP DOWN approach:

It is recalled that in this approach, DNL and INL tend to be observed and not extracted as in the BOTTOM UP approach. Thus, it is sufficient to communicate zero DNL and INL specifications to the DAC model to observe only the temperature impact. To cancel the temperature effect, we set T_MAX and T_MIN to 0. The simulation results are identical to those obtained in the section 4.3 and 4.2.

⇨ SystemC AMS allows the modélisation of generic non-ideal models and testbenches that cancel some non-idealities and test some others.

5 USE OF SYSTEMC AMS IN CADENCE ENVIRONMENT

So far, all the models developed have been simulated in the Eclipse environment. The purpose of this section is to see the results of these models when simulated in a CAD environment.

The first step is to prepare the environment. I will use the Xcelium which is a logic parallel simulator. In the second step, I will show the obtained results obtained in SimVision. And in this part, I will focus on the results of the generic models. That means, the ideal DAC in the SENT and RATIOMETRIC mode, the non-ideal DAC for the two approaches proposed in the SENT mode (for sake of simplicity) and which consider the DNL and INL impact first and the temperature variation secondly.

5.1 Setting up the Cadence environment

This part was carried out with the help of Cadence that provided instructions for installing the SystemC AMS library for the Xcelium simulator. Details about the installation instructions are available in annex 26.

The developed models were therefore also simulated and validated with the Cadence Xcelium tool available on the EPFL servers, but it was not possible to do so on the Melexis servers because of the difficulties encountered in installing the SystemC AMS libraries and the complex Cadence simulation environment requiring the intervention of the Melexis CAD team. Once these problems resolved, the models presented should be able to be simulated on Melexis servers.

5.2 Simulation Results

we import the models' files already developed in the Eclipse environment to the Xcelium environment and simulate them using the following instructions (see annex 26):

xrun_s.csh *.cpp or xrun_s.csh *.cpp

This work done for the Ideal DAC model and the non-ideal DAC model in both approaches (TOP DOWN and BOTTOM UP) that include the temperature variation as well as the INL and DNL effect.

Vout is the DAC output signal and Voutampl is the amplified output signal (Gain = 2).

- Ideal DAC in SENT mode:

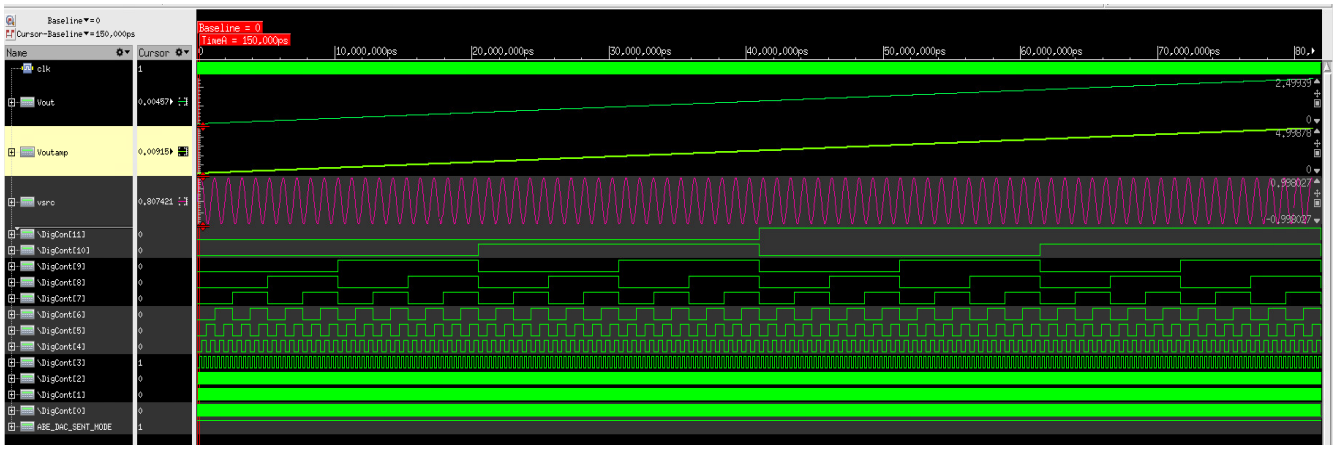

Figure 30: The SimVision simulation results of the ideal DAC operation in the SENT mode

- Ideal DAC in RATIOMETRIC mode:

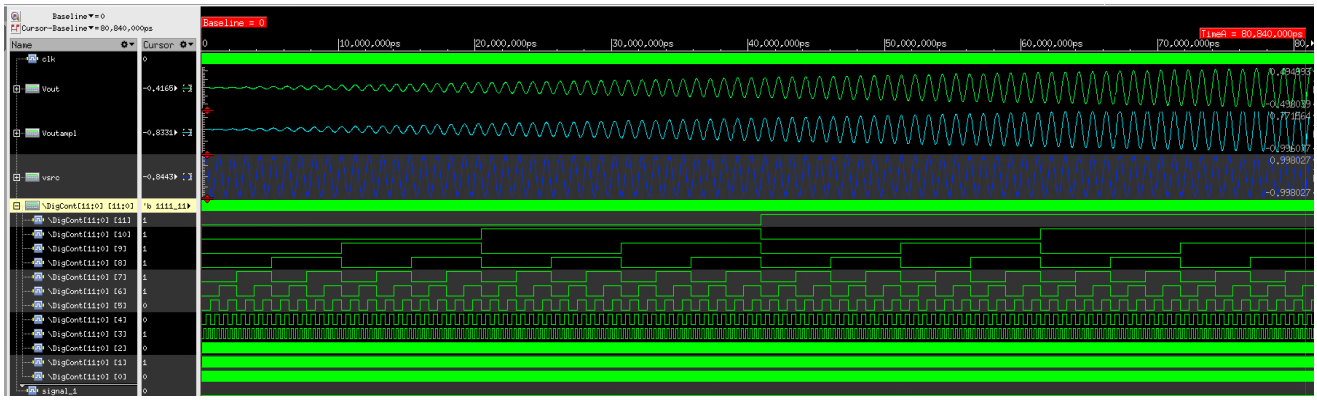

Figure 31: The SimVision simulation results of the ideal DAC operation in the RATIO-METRIC mode

- TOP DOWN Approach INL and DNL observation:

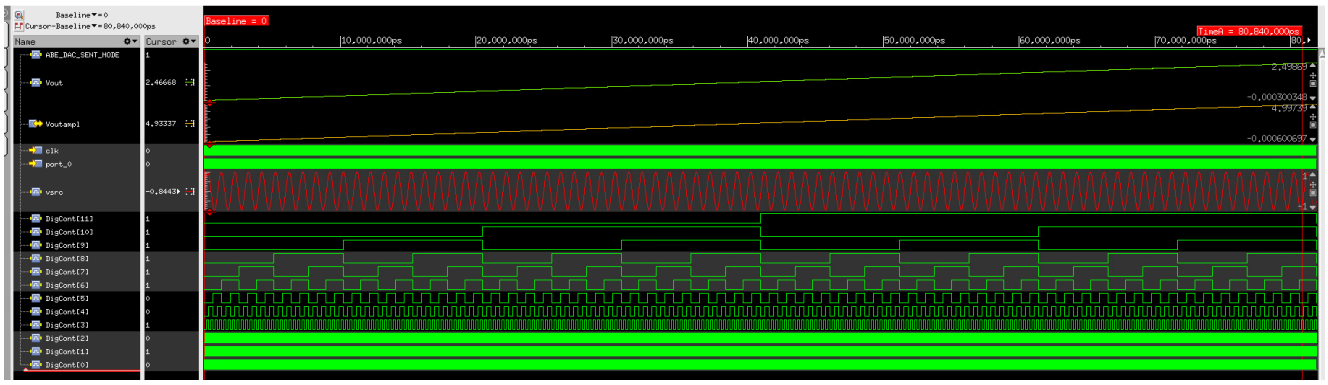

Figure 32: The SimVision simulation results of the non-ideal DAC in the TOP DOWN Approach in SENT mode for INL and DNL observation

- TOP DOWN Approach temperature observation:

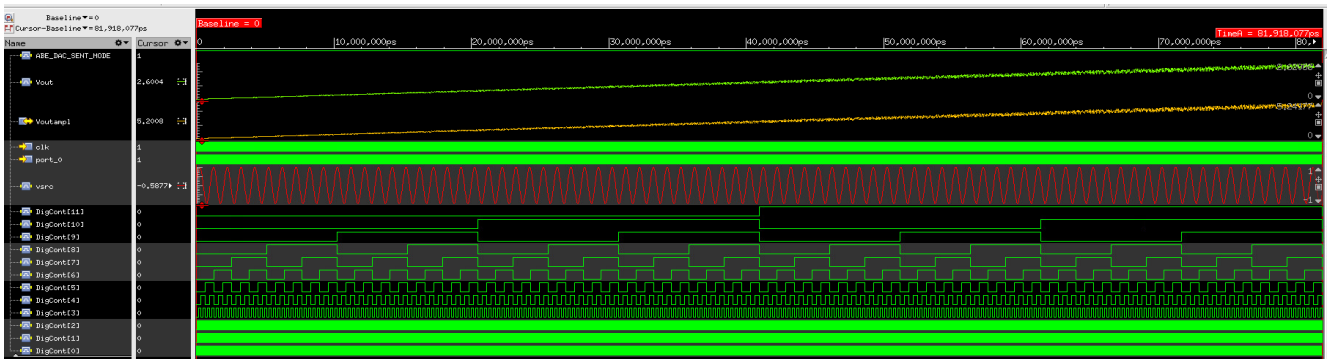

Figure 33: The SimVision simulation results of the non-ideal DAC in the TOP DOWN Approach in SENT mode for temperature observation

- BOTTOM UP Approach INL and DNL observation:

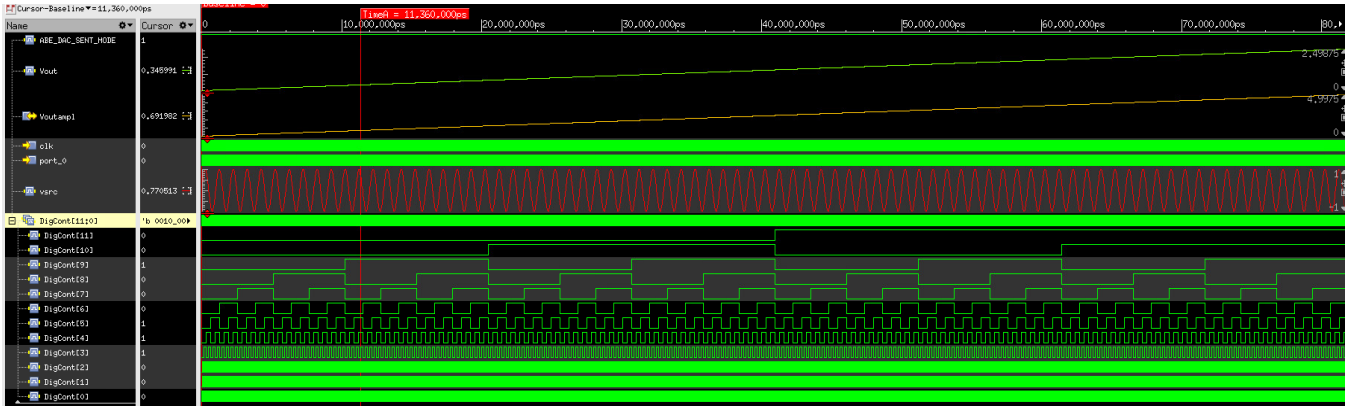

Figure 34: The SimVision simulation results of the non-ideal DAC in the BOTTOM UP Approach in SENT mode for INL and DNL observation

- BOTTOM UP Approach temperature observation:

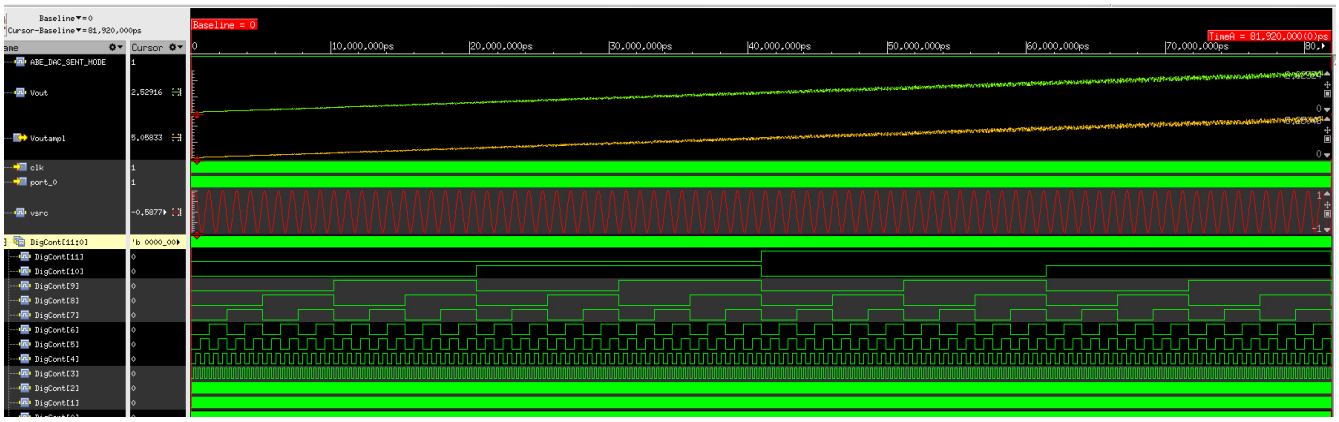

Figure 35: The SimVision simulation results of the non-ideal DAC in the BOTTOM UP Approach in SENT mode for temperature observation

We obtain curves similar to those already obtained when simulated in the Eclipse environment. However, SimVision offers more options and is more structured. As an example, the Digital Control signal can be split into 12 digital signals for each bit.

⇨ These results confirm again the modules designed using SystemC and SystemC AMS.

6 CONCLUSION

This internship was a challenge to see up to where I can convince Melexis of the advantages and capabilities of SystemC AMS and whether it is a good alternative to the Verilog AMS and can be used later in the verification flow for their future products or not.

Indeed, SystemC AMS allowed to model in a concrete and efficient way ideal behaviors of electronic components (PLL and DAC) and the temperature dependency of the DAC without adding any external port. This modeling is more complicated if the Verilog AMS is used. This work has also highlighted the capabilities of SystemC AMS to implement complementary approaches to describe the INL and DNL effect (BOTTOM UP and TOP DOWN) and to develop generic models (ideal DAC and non-ideal DAC) and different testbenches that allow to visualize some concepts included in the same model and cancelling others. These models were confirmed in two different modeling environments (Eclipse and Cadence).

The SystemC AMS which is based on discrete-time computation allows to reduce the simulation time and to minimize the energy consumption comparing to Verilog AMS which uses the event-driven formalism while giving results in accordance with the theory.

7 PERSONAL COMMENTS

7.1 Personal conclusion

At the time of the assessment, I will say that my master thesis project was a learning experience. I had the chance to learn new programming languages and to deepen my knowledge

in the field of modeling and design. My tutors listened to me despite the exceptional health situation and encouraged my proposals and ideas.

During my internship, I went through many difficulties. The epidemic circumstances prevent me to work on the site and get advantage of the presence of the mixed signal team who might have helped me improve my knowledge more. Moreover, despite my experience acquired in C++, I found that SystemC and SystemC AMS use much deeper and specific concepts and it took me time to get used to it. Besides, the major problem encountered during the internship is the setting up and the installation of the work environment in Cadence. In fact, Melexis environment is set up in a way to integrate Verilog and Verilog AMS models and does not contain the SystemC AMS libraries necessary to test my models. Therefore, we asked Cadence to help us and they gave us instructions to make the installation when they were available. After many trials, we succeeded to get simulation result using the Xcelium simulator.

To summarize, I really enjoyed the work I did during the last 6 months and this internship was for me a confirmation of my choice of career.

7.2 Acknowledgments

I dedicate this work to my two dear parents, Mohamed and Ikram, that no dedication can express my sincere gratitude for their unconditional love, limitless patience and continued support. Thank you for teaching me to always aim very high, higher than I consider myself capable of reaching and to always remember to look up at the stars and not down at my feet. All those days when I was determined, persevering and dreaming, I owe it all to you, Mom and Dad.

I would like to thank Mr. Alain VACHOUX, for having accepted to supervise me and for having supported me throughout this master thesis. I thank him for his presence as well as his help in solving my problems while keeping me autonomous.

I thank Mr. Alessandro BASILI and Mr. Michele PORTOLAN for their regular assistance during my internship.

I would like also to thank Melexis for affording all the necessary equipment so that I can work from home without any difficulty.

8 EXPECTED GANTT CHART

The GANTT Chart presents the planning of the work. Each color reflects a different step in the master thesis.

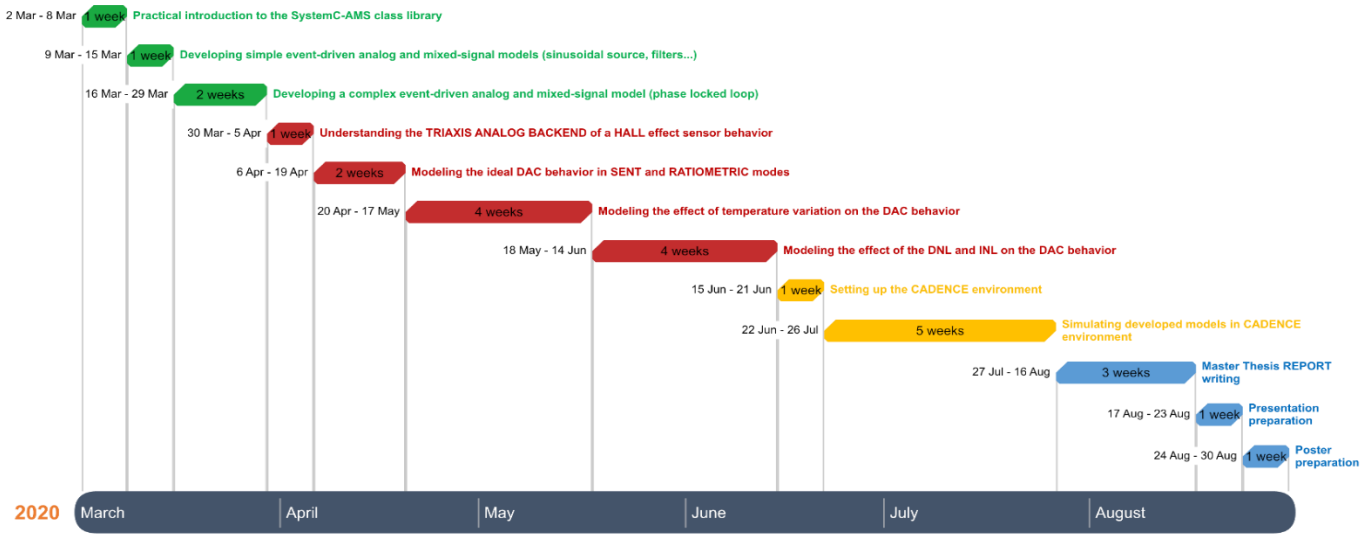

Figure 36: Expected GANTT Chart

9 ACTUAL GANTT CHART

The GANTT Chart presents the planning of the work. Each color reflects a different step in the master thesis.

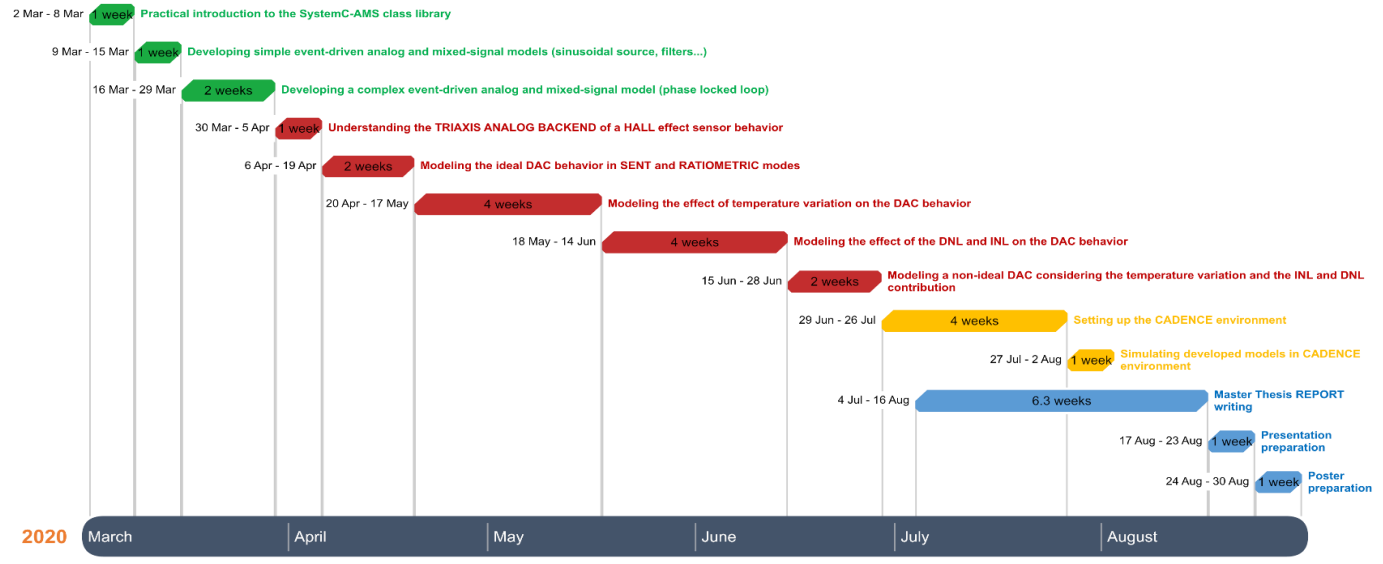

Figure 37: Actual GANTT Chart

10 BIBLIOGRAPHY

[1] Martin Barnasconi and Christoph Grimm, March 8 2010, SystemC AMS extensions User's Guide, https://www.accellera.org/images/downloads/standards/systemc/OSCI_SystemC_AMS_Users_Guide.pdf

[2] Gen III Triaxis® rotary and linear position sensor IC with PSI5 output, https://www.melexis.com/en/product/MLX90373/Triaxis-Rotary-Linear-Position-Sensor-PSI5-Output

[3] Melexis Products, https://www.melexis.com/en/products

[4] Melexis inspired Engineering, https://www.melexis.com/en/about-us

[5] Steve Arar, March 13 2019, What Are the DNL and INL Specifications of a DAC? Non-Linearity in Digital-to-Analog Converters, https://www.allaboutcircuits.com/technical-articles/understanding-dnl-and-inl-specifications-of-a-digital-to-analog-converter/

[6] Walt Kester, Basic DAC Architectures II: Binary DACs, https://www.analog.com/media/en/training-seminars/tutorials/MT-015.pdf

[7] James Bryant, Walt Kester, DATA CONVERTER ARCHITECTURES, https://www.analog.com/media/en/training-seminars/design-handbooks/Data-Conversion-Handbook/Chapter3.pdf

[8] Coefficient de température de résistance, https://riverglennapts.com/fr/resistance/748-temperature-coefficient-of-resistance.html

[9] Divyesh Gajjar, Analog Mixed Signal Verification Methodology (AMSVM), https://www.design-reuse.com/articles/28333/analog-mixed-signal-verification-methodology.html

[10] Linear Micro Author, August 16, 2019, COMBINING ANALOG AND DIGITAL ICS FOR MIXED SIGNAL ASIC DESIGN, https://linearmicrosystems.com/analog-digital-ics-asic-design/

[11] Steve Arar, March 19 2019, DNL and INL Specifications of a DAC: Interpreting INL Shape, https://www.allaboutcircuits.com/technical-articles/understanding-dnl-inl-specifications-digital-to-analog-converter/

[12] Rodrigo Cortés Porto, Ludovic Apvrille, François Pêcheux, 21 Oct 2019, A Tool for High-Level Modeling of Analog/Mixed Signal Embedded Systems Daniela Genius, https://hal.sorbonne-universite.fr/hal-01963837/document

11 ANNEXES

Annex 1: **Module that defines a sinusoidal source**

```
#ifndef SRC_SINESRC_H_
#define SRC_SINESRC_H_
#include <systemc-ams>
class Sinesrc: public sca_tdf::sca_module {
public:
   sca_tdf::sca_out<double> out;
   Sinesrc(const sc_core::sc_module_name& name,
           double ampl = 1.0,    // [-]
           double freq = 50.0,   // [Hz]
           double offs = 0.0,    // [-]
           double delay = 0.0,   // [s]
           double phase = 0.0,   // [deg]
           double theta = 0.0,   // [1/s]
           unsigned long rate = 1);
   virtual ~Sinesrc() {}

protected:
   void set_attributes();
   void initialize();
   void processing();
   void ac_processing();

private:
   double ampl_;     // amplitude [-]
   double omega_;    // angular frequency [rad/s]
   double offs_;     // DC offset [-]
   double delay_;    // oscillation start delay time [s]
   double phi_;      // phase delay [rad]
   double theta_;    // damping factor [1/s]
   unsigned long rate_;  // sample rate of output port
};

#endif /* SRC_SINESRC_H_ */
```

Annex 2: **Develop the sinusoidal source functions (set_attributes, initialize, processing, ac_processing)**

```
#include "Sinesrc.h"
#include <cmath>
#include <cassert>
#include <iostream>
Sinesrc::Sinesrc(const sc_core::sc_module_name& name,
                 double ampl,
                 double freq,
                 double offs,
                 double delay,
                 double phase,
                 double theta,
                 unsigned long rate)
: out("out"),
  ampl_(ampl),
  omega_(2.0*M_PI*freq),
  offs_(offs),
  delay_(delay),
  phi_((M_PI*phase)/180.0),
  theta_(theta),
  rate_(rate >= 1 ? rate : 1)
{}

void Sinesrc::set_attributes() {
   out.set_rate(rate_);
```

```
}

void Sinesrc::initialize() {
   using namespace std;
   assert(rate_ == out.get_rate());

#ifdef DEBUG
   cout << "--- "<< name() << " (Sinesrc)" << endl
        << "    offs  = " << offs_ << endl
        << "    ampl  = " << ampl_ << endl
        << "    freq  = " << omega_ /(2.0*M_PI) << " Hz" << endl
        << "    delay = " << delay_ << " s" << endl
        << "    theta = " << theta_ << " s^-1" << endl
        << "    phase = " << phi_*(180.0/M_PI) << " deg" << endl
        << "    rate  = " << rate_ << endl;
#endif // DEBUG
}

void Sinesrc::processing() {
   using namespace std;
   double t = out.get_time().to_seconds();
   double dt = out.get_timestep().to_seconds();

   for (unsigned long i = 0; i < rate_; ++i) {
      double v = offs_;
      if (t >= delay_) {
         v += ampl_*exp(-(t - delay_)*theta_)*sin(omega_*(t - delay_) +
          phi_);
     }
     out.write(v);
     t += dt;
   }
}
void Sinesrc::ac_processing() {
   sca_ac_analysis::sca_ac(out) = 1.0;
}
```

Annex 3: **Module that allows the generation of all the possible NBITS binary sequences in a successive way**

```
#include "systemc-ams.h"
template <int NBITS>
SC_MODULE(inputcombination) {
   sc_in<bool> clk;
   sc_out<sc_bv<NBITS> > output;
   void gen() {
      int v = 0;
      while (true) {
         output = v;
         v += 1;
         wait();
      }
   }
   SC_CTOR(inputcombination) {
      SC_THREAD(gen);
      sensitive << clk.pos();
   }
};
```

Annex 4: **Module that describes the behavior of an amplifier of Gain 2**

```
#ifndef SRC_AMPLIFICATION_H_
#define SRC_AMPLIFICATION_H_
#include "systemc-ams.h
class Amplification: public sca_tdf::sca_module {
public:
```

```
    sca_tdf::sca_in<double>  in; // port input
    sca_tdf::sca_out<double> out; //port output

    Amplification(const sc_core::sc_module_name& name,
                                double Gain = 2.0)
    : Gain_(Gain)
        {}
    virtual ~Amplification() {}

protected:
   void initialize(){
#ifdef DEBUG
      using namespace std;
      cout << "-- " << name() << " (AMPLIFICATION)" << endl
           << "   Gain_ = " << Gain_  << endl;
#endif
}
   void processing()  {
      out.write(Gain_*in.read());
}

private:
   double Gain_; // AOP Gain
};
#endif /* SRC_AMPLIFICATION_H_ */
```

Annex 6: Implementing the DAC operation in the ideal case showing the SENT mode and the RATIOMETRIC mode

```
#ifndef SRC_DAC_IDEAL_H_
#define SRC_DAC_IDEAL_H_
#include "systemc-ams.h"
template <int NBITS> // NBITS digital control bits number

class Dac: public sca_tdf::sca_module {
public:

    sca_tdf::sca_de::sca_in<sc_bv<NBITS> > DigCont; // port input digital control
    sca_tdf::sca_de::sca_in<bool> ABE_DAC_SENT_MODE ; // ABE_DAC_SENT_MODE Signal
    sca_tdf::sca_out<double> out; //port output
    sca_tdf::sca_in<double> Vs; // supply voltage signal "sinusoidal source"

    Dac(const sc_core::sc_module_name& name,
                          double Vsint1 = 2.5)
    :  Vsint1_(Vsint1)
        {}
    virtual ~Dac() {}
protected:
    void initialize(){
    using namespace std;
#ifdef DEBUG
     cout << "-- " << name() << " (Dac)" << endl
          << "   Vsint1  = " << Vsint1_ << "V" << endl;
#endif
}
    void processing(){
        double sum = 0.0 ; //sum of 2^i where i is the index of bits having the
value1
        for (int i = 0 ; i < NBITS; i++) {
                if (DigCont.read().get_bit(i) == 1 ) sum += pow(2.0,i);

                          }
        if (ABE_DAC_SENT_MODE == true)
                out.write(sum*(Vsint1_/pow(2.0,NBITS))); // sent mode
        else
                out.write(sum*(Vs.read()/pow(2.0,NBITS+1))); // ratiometric mode
```

```
                }
Private:
        double Vsint1_; // Internal Supply Voltage 1 [V]
};
#endif /* SRC_DAC_IDEAL_H_ */
```

Annex 7: Testbench to simulate the ideal DAC in the SENT mode

```
#include "Amplification.h"
#include "Dac_ideal.h"
#include "Sinesrc.h"
#include "Successive_generation_NBITSsequences.h"
int sc_main(int argc, char* argv[]) {
        using namespace sc_core;
        using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;

        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters

    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
    const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals

        sca_tdf::sca_signal<double> Vout; // DAC output
        sca_tdf::sca_signal<double> Voutampl; // Amplification output
        sca_tdf::sca_signal<double> vsrc; // sinusoidal source
        sc_signal<sc_bv<NBITS> > DigCont; // Digital Control signal
        sc_signal<bool> ABE_DAC_SENT_MODE; // ABE DAC SENT MODE signal

        // Dac components
        ABE_DAC_SENT_MODE.write(true);
        Sinesrc i_ref("i_ref", AMPL, FREQ, OFFS);
        i_ref.out(vsrc);
        i_ref.set_timestep(TSTEP);
        inputcombination<NBITS> i_comb("i_comb");
        i_comb.clk(clk);
        i_comb.output(DigCont);
        Dac<NBITS> i_dac("i_dac",VSINT1);
        i_dac.Vs(vsrc);
        i_dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
        i_dac.DigCont(DigCont);
        i_dac.out(Vout);
        i_dac.set_timestep(TSTEP);
        Amplification i_amp ("i_amp",GAIN);
        i_amp.in(Vout);
        i_amp.out(Voutampl);
        i_amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_ideal_SENT_MODE_tb");
         sca_util::sca_trace(tfp, DigCont, "DigCont");
         sca_util::sca_trace(tfp, Vout, "Vout");
```

```
          sca_util::sca_trace(tfp, Voutampl, "Voutampl");
          sca_util::sca_trace(tfp,ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
          sca_util::sca_trace(tfp,vsrc , "vsrc");

          // simulation
          sc_start(TSTOP);
          return 0;
}
```

Annex 8: **Testbench to simulate the ideal DAC in RATIOMETRIC mode**

```
#include "Amplification.h"
#include "Dac_ideal.h"
#include "Sinesrc.h"
#include "Successive_generation_NBITSsequences.h"

int sc_main(int argc, char* argv[]) {
         using namespace sc_core;
        using namespace sca_core;
         using namespace sca_util;

         // system parameters
         const int NBITS = 12;
         const double VSINT1 = 2.5; // [V]
         const double GAIN = 2.0;

         // source parameters
         const double AMPL = 1.0;
         const double OFFS = 0;
         const double FREQ = 1.0e6;

         // simulation parameters

    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
    const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

         // signals

         sca_tdf::sca_signal<double> Vout; // DAC output
         sca_tdf::sca_signal<double> Voutampl; // Amplification output
         sca_tdf::sca_signal<double> vsrc; // sinusoidal source
         sc_signal<sc_bv<NBITS> > DigCont; // Digital Control signal
         sc_signal<bool> ABE_DAC_SENT_MODE; // ABE DAC SENT MODE signal

         // Dac components
         ABE_DAC_SENT_MODE.write(false);
         Sinesrc i_ref("i_ref", AMPL, FREQ, OFFS);
         i_ref.out(vsrc);
         i_ref.set_timestep(TSTEP);
         inputcombination<NBITS> i_comb("i_comb");
         i_comb.clk(clk);
         i_comb.output(DigCont);
         Dac<NBITS> i_dac("i_dac",VSINT1);
         i_dac.Vs(vsrc);
         i_dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
         i_dac.DigCont(DigCont);
         i_dac.out(Vout);
         i_dac.set_timestep(TSTEP);
         Amplification i_amp ("i_amp",GAIN);
         i_amp.in(Vout);
         i_amp.out(Voutampl);
         i_amp.set_timestep(TSTEP);

          // tracing
```

```
          sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_ideal_RATIOMETRIC_MODE_tb");
          sca_util::sca_trace(tfp, DigCont, "DigCont");
         sca_util::sca_trace(tfp, Vout, "Vout");
          sca_util::sca_trace(tfp, Voutampl, "Voutampl");
          sca_util::sca_trace(tfp,ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
          sca_util::sca_trace(tfp,vsrc , "vsrc");

          // simulation
          sc_start(TSTOP);
          return 0;
}
```

Annex 9: **Module describing the DAC behavior considering the temperature variation**

```
#ifndef SRC_DAC_TEMP_H_
#define SRC_DAC_TEMP_H_
#include "systemc-ams.h"
#include <cstring>
#include "scams/predefined_moc/tdf/sca_tdf_sc_in.h"
template <int NBITS> // NBITS digital control bits number

class Dac_temp: public sca_tdf::sca_module {
public:

        sca_tdf::sca_de::sca_in<sc_bv<NBITS> > DigCont; // port input digital
control
        sca_tdf::sca_de::sca_in<bool> ABE_DAC_SENT_MODE ; // ABE_DAC_SENT_MODE
Signal
         sca_tdf::sca_out<double> out; //port output
         sca_tdf::sca_in<double> Vs; // supply voltage signal

         void set_Temp(double T)
         { temp = T;}

         Dac_temp(const sc_core::sc_module_name& name,
                                    double Vsint1 = 2.5,
                                    double T_amb = 35.0)

                              :  Vsint1_(Vsint1), T_amb_(T_amb)
         {}
         virtual ~Dac_temp() {}
protected:
         void initialize(){
#ifdef DEBUG
       using namespace std;
       cout << "-- " << name() << " (Dac)" << endl
            << "   Vs   = " << Vs_ << " V" << endl
           << "   Vsint1   = " << Vsint1_ << "V" << endl
       }
#endif
}
 void processing(){

                  double inter;
                  double sum = 0.0 ;
                           for (int i = 0 ; i < NBITS; i++) {
                                    if (DigCont.read().get_bit(i) == 1 ) sum +=
pow(2.0,i);

                           }
                           if (ABE_DAC_SENT_MODE == true)
                                    inter = (sum*(Vsint1_/pow(2.0,NBITS)));
                           else
                                    inter = (sum*(Vs.read()/pow(2.0,NBITS+1)));

                              out.write (inter-inter*(temp - T_amb_)/ T_amb_);
```

```
                        using namespace std;
                        cout << "-- " << temp <<" temp " << endl;

        }
private:
        double Vsint1_; // Internal Supply Voltage 1 [V]
        double T_amb_; // temperature ambiante
        double temp;
};
#endif /* SRC_DAC_TEMP_H_ */
```

Annex 10: **Module that contains the update_temperature function, all the testbench instruction (mapping) and sensible to the clock position in order to vary the temperature at each clock cycle, simultaneously with the processing function (this model is developed for the SENT mode)**

```
#ifndef SRC_TEMP_VAR_SENT_MODE_H_
#define SRC_TEMP_VAR_SENT_MODE_H_
#include "Successive_generation_NBITSsequences.h"
#include "Amplification.h"
#include "Dac_temp.h"
#include "Sinesrc.h"
#include "systemc-ams.h"
#include <cstdlib> // for std::rand
template <int NBITS> // NBITS digital control bits number
class tempvariation: public sc_module {
public:

        sc_in<bool> clk;
        sc_in<bool> clk_temp;
        sca_tdf::sca_signal<double> Vout;
        sca_tdf::sca_out<double> Voutampl;
        sca_tdf::sca_signal<double> vsrc;
        sc_signal<bool> ABE_DAC_SENT_MODE;;
        sc_signal<sc_bv<NBITS> > DigCont;
        Sinesrc Sin;
        Dac_temp<NBITS> Dac;
        inputcombination<NBITS> digital;
        Amplification amp;
        void temperature_update () {

                double T;
                double x;
                while (true)
                 {
                x =  ((double)std::rand())/RAND_MAX ;
                T = T_MIN_ + x * (T_MAX_-T_MIN_);
                Dac.set_Temp(T);
                wait();
                using namespace std;
                cout << "-- " << T << " T " << endl;
                 }
            }
         SC_HAS_PROCESS(tempvariation);

        tempvariation(const sc_core::sc_module_name& name,
                                double VSINT1 = 2.5 , // [V]
                        double GAIN = 2.0,
                        double T_AMB = 37.0,
                        double AMPL = 1.0,
                        double OFFS = 0.0,
                        double FREQ = 1.0e6,
                        double T_MAX = 40.0,
                        double T_MIN = 35.0)
```

```
          :  VSINT1_(VSINT1), GAIN_(GAIN), T_AMB_(T_AMB),AMPL_(AMPL),
                OFFS_(OFFS),FREQ_(FREQ),T_MAX_(T_MAX),T_MIN_(T_MIN),
                clk ("clk"), Vout("Vout"), Voutampl("Voutampl"),vsrc("vsrc"),
                DigCont("DigCont"),ABE_DAC_SENT_MODE("ABE_DAC_SENT_MODE"),
                Sin("Sin",AMPL, FREQ,
OFFS),Dac("Dac",VSINT1,T_AMB),digital("digital"),
                amp("amp")

        {
                ABE_DAC_SENT_MODE.write(true);
        // sin component
                Sin.out(vsrc);
        // Digital control component
                digital.clk(clk);
                digital.output(DigCont);
        // DAC component
                Dac.Vs(vsrc);
                Dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
                Dac.DigCont(DigCont);
                Dac.out(Vout);
        // amplification component
                amp.in(Vout);
                amp.out(Voutampl);

                SC_THREAD(temperature_update);
                sensitive << clk.pos();
}

        virtual ~tempvariation() {}

protected:
private:
             double VSINT1_ ; // [V]
             double GAIN_ ;
             double T_AMB_ ;
             double AMPL_ ;
             double OFFS_ ;
             double FREQ_ ;
             double T_MAX_ ;
             double T_MIN_ ;
};

#endif /* SRC_DAC_TEMP_H_ */
```

Annex 11: **Module that contains the update_temperature function, all the testbench instruction (mapping) and sensible to the clock position in order to vary the temperature at each clock cycle, simultaneously with the processing function (this model is developed for the RATIOMETRIC mode)**

```
#ifndef SRC_TEMP_VAR__RATIOMETRIC_MODE_H_
#define SRC_TEMP_VAR__RATIOMETRIC_MODE_H_
#include "Successive_generation_NBITSsequences.h"
#include "Amplification.h"
#include "Dac_temp.h"
#include "Sinesrc.h"
#include "systemc-ams.h"
#include <cstdlib> // for std::rand
template <int NBITS> // NBITS digital control bits number
class tempvariation: public sc_module {
public:

        sc_in<bool> clk;
        sc_in<bool> clk_temp;
        sca_tdf::sca_signal<double> Vout;
        sca_tdf::sca_out<double> Voutampl;
```

```
        sca_tdf::sca_signal<double> vsrc;
        sc_signal<bool> ABE_DAC_SENT_MODE;;
        sc_signal<sc_bv<NBITS> > DigCont;

        Sinesrc Sin;
        Dac_temp<NBITS> Dac;
        inputcombination<NBITS> digital;
        Amplification amp;
         void temperature_update () {

                double T;
                double x;
                while (true)
                 {
                x =  ((double)std::rand())/RAND_MAX ;
                T = T_MIN_ + x * (T_MAX_-T_MIN_);
                Dac.set_Temp(T);
                wait();
                using namespace std;
                cout << "-- " << T << " T " << endl;
                 }
           }
         SC_HAS_PROCESS(tempvariation);

        tempvariation(const sc_core::sc_module_name& name,
                                double VSINT1 = 2.5 , // [V]
                            double GAIN = 2.0,
                            double T_AMB = 37.0,
                            double AMPL = 1.0,
                            double OFFS = 0.0,
                            double FREQ = 1.0e6,
                            double T_MAX = 40.0,
                            double T_MIN = 35.0)

          :  VSINT1_(VSINT1), GAIN_(GAIN), T_AMB_(T_AMB),AMPL_(AMPL),
                 OFFS_(OFFS),FREQ_(FREQ),T_MAX_(T_MAX),T_MIN_(T_MIN),
                 clk ("clk"), Vout("Vout"), Voutampl("Voutampl"),vsrc("vsrc"),
                 DigCont("DigCont"),ABE_DAC_SENT_MODE("ABE_DAC_SENT_MODE"),
                 Sin("Sin",AMPL, FREQ,
OFFS),Dac("Dac",VSINT1,T_AMB),digital("digital"),
                 amp("amp")

        {
                ABE_DAC_SENT_MODE.write(false);
        // sin component
                Sin.out(vsrc);
        // Digital control component
                digital.clk(clk);
                digital.output(DigCont);
        // DAC component
               Dac.Vs(vsrc);
               Dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
               Dac.DigCont(DigCont);
               Dac.out(Vout);
        // amplification component
               amp.in(Vout);
               amp.out(Voutampl);

               SC_THREAD(temperature_update);
               sensitive << clk.pos();
}

        virtual ~tempvariation() {}

protected:
```

```
private:
            double VSINT1_ ; // [V]
            double GAIN_ ;
            double T_AMB_ ;
            double AMPL_ ;
            double OFFS_ ;
            double FREQ_ ;
            double T_MAX_ ;
            double T_MIN_ ;
};

#endif /* SRC_DAC_TEMP_H_ */
```

Annex 12: **Testbench to simulate the DAC behavior considering the temperature variation**

```
#include "Amplification.h"
#include "Dac_temp.h"
#include "Sinesrc.h"
#include "systemc-ams.h"
#include <cstdlib> // for std::rand
#include "temp_var_sent_mode.h"

int sc_main(int argc, char* argv[]) {

        using namespace sc_core;
        using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;
        const double T_AMB = 37.0;
        const double T_MAX = 40.0;
        const double T_MIN = 35.0;

        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters

        const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
        const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
        const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
        sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
        sc_clock clk_temp("clk_temp", CLK_PER, 0.5, SC_ZERO_TIME, false);
        const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals

        sca_tdf::sca_signal<double> Voutampl;
        tempvariation<NBITS> i_test
        ("i_test",VSINT1,GAIN,T_AMB,AMPL,OFFS,FREQ,T_MAX,T_MIN);
        i_test.clk_temp(clk_temp);
        i_test.clk(clk);
        i_test.Voutampl(Voutampl);
        i_test.Dac.set_timestep(TSTEP);
        i_test.Sin.set_timestep(TSTEP);
        i_test.amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
        sca_util::sca_create_vcd_trace_file("dac_temp_variation_SENTMODE_tb");
         sca_util::sca_trace(tfp, i_test.DigCont, "DigCont");
```

```
          sca_util::sca_trace(tfp, i_test.Vout, "Vout");
          sca_util::sca_trace(tfp, Voutampl, "Voutampl");
          sca_util::sca_trace(tfp,i_test.ABE_DAC_SENT_MODE ,
                 "ABE_DAC_SENT_MODE");
          sca_util::sca_trace(tfp,i_test.vsrc , "vsrc");

          // simulation
          sc_start(TSTOP);
          return 0;
}
```

Annex 13: **Module describing the effect of the INL and DNL on the DAC operation in a TOP DOWN approach and valid only for a ramp Digital Control signal**

```
#ifndef SRC_DAC_DNL_INL_RAMPE_H_
#define SRC_DAC_DNL_INL_RAMPE_H_
#include "systemc-ams.h"
#include <cstdlib> // for std::rand
#include <time.h>
#include <stdio.h>
#include<stdlib.h>

template <int NBITS> // NBITS digital control bits number
class Dac_DNL_INL: public sca_tdf::sca_module {
public:

         double get_dnl() {
          return (current_dnl); }
          sca_tdf::sca_de::sca_in<sc_bv<NBITS> > DigCont; // port input digital
          sca_tdf::sca_de::sca_in<bool> ABE_DAC_SENT_MODE ; // ABE_DAC_SENT_MODE
Signal
         sca_tdf::sca_out<double> out; //port output
         sca_tdf::sca_in<double> Vs; // supply voltage signal
         sca_tdf::sca_trace_variable<double> current_dnl;

         Dac_DNL_INL(const sc_core::sc_module_name& name,
                                    double Vsint1 = 2.5,
                                    double DNL = 1.5,
                                    double INL = 2.0)
                              :  Vsint1_(Vsint1), DNL_(DNL), INL_(INL)
                                        , previous_output(0.0),  DNL_var(0.0),
INL_var(0.0)

         {}
         virtual ~Dac_DNL_INL() {}
protected:
         void initialize(){
         srand(999);
         using namespace std;

#ifdef DEBUG
      cout << "-- " << name() << " (Dac)" << endl
           << "   Vs  = " << Vs_ << " V" << endl
           << "   Vsint1  = " << Vsint1_ << "V" << endl;
#endif

 }
         void processing(){

             double x;
             double y;
             double sum = 0.0;
             double output_value = 0.0;
                 for (int i = 0 ; i < NBITS; i++) {
                         if (DigCont.read().get_bit(i) == 1 ) {
                                 sum += pow(2.0,i) ;}
```

```
                }
                do {
                        x =  ((double)std::rand())/RAND_MAX ; // double between 0
and 1
                        y = -INL_ + x * (INL_ - (-INL_));
                        output_value = sum + y;
                        current_dnl = previous_output - output_value + 1 ; }

             while (abs (current_dnl)> DNL_);


        if ( abs (y) >INL_var_) INL_var_ = abs (y);
        if (abs (current_dnl) >DNL_var_) DNL_var_ = abs(current_dnl);

         using namespace std;
        cout << "-- " << current_dnl << " current_dnl " << endl;
        cout << "-- " << y << " variation around ideal value " << endl;
        cout << "-- " << DNL_var_ << " maximum value of DNL till now " << endl;
        cout << "-- " << INL_var_ << " maximum value of INL till now " << endl;


                if (ABE_DAC_SENT_MODE == true)
                        out.write((output_value)*(Vsint1_/pow(2.0,NBITS)));
                else
                        out.write((output_value)*(Vs.read()/pow(2.0,NBITS+1)));

                previous_output = output_value;
        }

private:
        double Vsint1_; // Internal Supply Voltage 1 [V]
        double DNL_;
        double INL_;
        double previous_output; // valeur de dnl precedente;
        double DNL_var_ ;
        double INL_var_ ;
};
#endif /* SRC_DAC_DNL_INL_RAMPE_H_ */
```

Annex 14: **Testbench to simulate the module describing the effect of the INL and DNL on the DAC operation in a TOP DOWN approach and valid only for a ramp Digital Control signal**

```
#include "Amplification.h"
#include "Dac_DNL_INL_RAMPE.h"
#include "Sinesrc.h"
#include "Successive_generation_NBITSsequences.h"
int sc_main(int argc, char* argv[]) {
        using namespace sc_core;
         using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;
        const double DNL = 1.5; // [V]
        const double INL = 2.0;

        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters
```

```
    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
    sc_clock clk_DAC("clk_DAC", CLK_PER, 0.5, SC_ZERO_TIME, false);
        const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals

        sca_tdf::sca_signal<double> Vout;
        sca_tdf::sca_signal<double> Voutampl;
        sca_tdf::sca_signal<double> vsrc;
        sc_signal<sc_bv<NBITS> > DigCont;
        sc_signal<bool> ABE_DAC_SENT_MODE;

        // Dac components

        ABE_DAC_SENT_MODE.write(false);
        Sinesrc i_ref("i_ref", AMPL, FREQ, OFFS);
        i_ref.out(vsrc);
        i_ref.set_timestep(TSTEP);
        inputcombination<NBITS> i_comb("i_comb");
        i_comb.clk(clk);
        i_comb.output(DigCont);
        Dac_DNL_INL<NBITS> i_dac("i_dac",VSINT1,DNL,INL);
        i_dac.Vs(vsrc);
        i_dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
        i_dac.DigCont(DigCont);
        i_dac.out(Vout);
        i_dac.set_timestep(TSTEP);
        Amplification i_amp ("i_amp",GAIN);
        i_amp.in(Vout);
        i_amp.out(Voutampl);
        i_amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_DNL_INL_tb");
         sca_util::sca_trace(tfp, DigCont, "DigCont");
         sca_util::sca_trace(tfp, i_dac.current_dnl, "dnl_curve");
        sca_util::sca_trace(tfp, Vout, "Vout");
         sca_util::sca_trace(tfp, Voutampl, "Voutampl");
         sca_util::sca_trace(tfp,ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
         sca_util::sca_trace(tfp,vsrc , "vsrc");

         // simulation
         sc_start(TSTOP);
         return 0;
}
```

Annex15: **Module describing the effect of the INL and DNL on the DAC operation in a BOTTOM UP approach**

```
#ifndef SRC_DAC_INL_DNL_BOTTOM_UP_H_
#define SRC_DAC_INL_DNL_BOTTOM_UP_H_
#include "systemc-ams.h"
#include <cstdlib> // for std::rand
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
template <int NBITS> // NBITS digital control bits number

class Dac_DNL_INL: public sca_tdf::sca_module {
public:
```

```
    sca_tdf::sca_de::sca_in<sc_bv<NBITS> > DigCont; // port input digital control
"12 bits signal"
    sca_tdf::sca_de::sca_in<bool> ABE_DAC_SENT_MODE ; // ABE_DAC_SENT_MODE Signal
        sca_tdf::sca_out<double> out; //port output
        sca_tdf::sca_in<double> Vs; // supply voltage signal
        sca_tdf::sca_trace_variable<double> dnl_variation;
        sca_tdf::sca_trace_variable<double> inl_variation;


        Dac_DNL_INL(const sc_core::sc_module_name& name,
                                    double Vsint1 = 2.5,
                                    double var_max = 0.37,
                                    double var_min = -0.37)
                            :  Vsint1_(Vsint1), var_max_(var_max),DNL_(0.0),

var_min_(var_min),INL_(0.0),previous_output(0.0),
                                            weight_bit(NBITS),tab_ref_ideal((int)
pow(2.0,NBITS))
        {}

        virtual ~Dac_DNL_INL() {}
protected:
        void initialize() {

                // function to guarantee the generation of random values
                srand(999);
                double var_rand; // random value between 0 and 1
                double var =0.0;

                for (int j = 0 ; j < NBITS; j++) {
                        var_rand = ((double) rand()/RAND_MAX);
                        var = -0.37 + var_rand*0.74;
                        weight_bit[j] = pow(2.0,j) + var ;}

                for (int i = 0 ; i < (int)(pow(2.0,NBITS)); i++) {

                    sc_bv<NBITS> x;
                        x = (sc_uint<NBITS>)(i); // coder l'entier i sur NBITS

                        double sum_ideal = 0.0;
                        for (int j = 0 ; j < NBITS; j++) {
                                if (x[j] == 1 ) {
                                                sum_ideal += pow(2.0,j) ;}
                                }
                        tab_ref_ideal[i]=sum_ideal;}
            using namespace std;
#ifdef DEBUG
      cout << "-- " << name() << " (Dac)" << endl
           << "   Vs  = " << Vs_ << " V" << endl
           << "   Vsint1  = " << Vsint1_ << "V" << endl
                << "   DNL = " << DNL_ << "Lsb" << endl
                << "   INL = " << INL_ << "Lsb" << endl;

#endif
}
        void processing(){

                double output = 0.0;
                double sum = 0.0 ;
                for (int i = 0 ; i < NBITS; i++) {
                        if ((sc_uint<NBITS>)DigCont.read().get_bit(i) == 1 ) {
                        sum += weight_bit[i] ;}
                                }

                output = sum ;
                if (ABE_DAC_SENT_MODE == true)
```

```
                        out.write(output*(Vsint1_/pow(2.0,NBITS)));

                else
                        out.write(output*(Vs.read()/pow(2.0,NBITS+1)));

                if (sc_int<NBITS> (DigCont.read()) < 0){
                        dnl_variation = output - (1 + previous_output);
                        inl_variation = output - tab_ref_ideal[(sc_int<NBITS>)
(DigCont.read()) + (int) pow(2.0,NBITS)];}

                else
                         {dnl_variation = output - (1 + previous_output);
                          inl_variation = output -
tab_ref_ideal[(sc_int<NBITS>)(DigCont.read())];}

                if (abs (DNL_)< abs (dnl_variation)) DNL_ = dnl_variation;
                if (abs (INL_) < abs (inl_variation)) INL_ = inl_variation;


                        using namespace std;
                                cout << "-- " << "DNL" << DNL_ << endl
                                     << "-- " << "INL" << INL_ << endl;
                previous_output=output;

        }
private:
        double Vsint1_; // Internal Supply Voltage 1 [V]
        double var_max_; // maximum variation of the bits weights
        double var_min_; // minimum variation of the bits weights
        double DNL_;
        double INL_;
        double previous_output;
        std::vector<double> weight_bit; // table containing random bit weight
        std::vector<double> tab_ref_ideal; // table containing ideal output values

};

#endif /* SRC_DAC_DNL_INL_H_ */
```

Annex 16: **Testbench to simulate the module describing the effect of the INL and DNL of the DAC operation in a BOTTOM UP approach**

```
#include "Amplification.h"
#include "Dac_INL_DNL_BOTTOM_UP.h"
#include "Sinesrc.h"
#include "Successive_generation_NBITSsequences.h"
int sc_main(int argc, char* argv[]) {
        using namespace sc_core;
        using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;

        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters
    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
```

```
    sc_clock clk_DAC("clk_DAC",CLK_PER, 0.5, SC_ZERO_TIME, false);
    const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals

        sca_tdf::sca_signal<double> Vout;
        sca_tdf::sca_signal<double> Voutampl;
        sca_tdf::sca_signal<double> vsrc;
        sc_signal<sc_bv<NBITS> > DigCont;
        sc_signal<bool> ABE_DAC_SENT_MODE;

        ABE_DAC_SENT_MODE.write(true);
        // Dac components
        Sinesrc i_ref("i_ref", AMPL, FREQ, OFFS);
        i_ref.out(vsrc);
        i_ref.set_timestep(TSTEP);
         inputcombination<NBITS> i_comb("i_comb");
         i_comb.clk(clk);
         i_comb.output(DigCont);
        Dac_DNL_INL<NBITS> i_dac("i_dac",VSINT1);
        i_dac.Vs(vsrc);
        i_dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
        i_dac.DigCont(DigCont);
        i_dac.out(Vout);
        i_dac.set_timestep(TSTEP);
        Amplification i_amp ("i_amp",GAIN);
        i_amp.in(Vout);
        i_amp.out(Voutampl);
        i_amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_DNL_INL_BOTTOM_UP_tb");
         sca_util::sca_trace(tfp, DigCont, "DigCont");
        sca_util::sca_trace(tfp, Vout, "Vout");
         sca_util::sca_trace(tfp, Voutampl, "Voutampl");
         sca_util::sca_trace(tfp,ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
         sca_util::sca_trace(tfp,vsrc , "vsrc");
         sca_util::sca_trace(tfp,i_dac.dnl_variation , "dnl_variation");
         sca_util::sca_trace(tfp,i_dac.inl_variation , "inl_variation");

         // simulation
         sc_start(TSTOP);
         return 0;
}
```

Annex 17: **Module describing the effect of the INL and DNL on the DAC operation in a TOP DOWN approach valid for all possible input combination order**

```
#ifndef SRC_DAC_DNL_INL_TOP_DOWN_H_
#define SRC_DAC_DNL_INL_TOP_DOWN_H_

#include "systemc-ams.h"
#include <cstdlib> // for std::rand
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
template <int NBITS> // NBITS digital control bits number

class Dac_DNL_INL: public sca_tdf::sca_module {
public:

    sca_tdf::sca_de::sca_in<sc_bv<NBITS> > DigCont; // port input digital control
"12 bits signal"
    sca_tdf::sca_de::sca_in<bool> ABE_DAC_SENT_MODE ; // ABE_DAC_SENT_MODE Signal
        sca_tdf::sca_out<double> out; //port output
```

```
        sca_tdf::sca_in<double> Vs; // supply voltage signal
        sca_tdf::sca_trace_variable<double> current_dnl;


        Dac_DNL_INL(const sc_core::sc_module_name& name,
                                    double Vsint1 = 2.5,
                                    double DNL = 1.5,
                                    double INL = 2)
                            :  Vsint1_(Vsint1),
DNL_(DNL),INL_(INL),tab_ref_ideal((int)pow(2.0,NBITS)),tab_ref_not_ideal((int)pow(2
.0,NBITS))


        {}

        virtual ~Dac_DNL_INL() {}
protected:
        void initialize() {

                // function to guarantee the generation of random values
                srand(999);
                for (int i = 0 ; i < (int)(pow(2.0,NBITS)) ; i++) {

                    double var_rand; // random value between 0 and 1
                    double var;           //      random value between DNL and -
DNL
                    double difference = 0.0; // difference between ideal value and
not ideal value to conclude the INL
                    sc_bv<NBITS> x;
                        x = (sc_uint<NBITS>)(i); // coder l'entier i sur NBITS

                        // Reference table of ideal values

                        double sum_ideal = 0.0;
                        for (int j = 0 ; j < NBITS; j++) {
                                if (x[j] == 1 ) {
                                                sum_ideal += pow(2.0,j) ;}
                                }
                        tab_ref_ideal[i]=sum_ideal;

                        // Reference table of not ideal values respecting DNL and
INL

                        if (i==0) { tab_ref_not_ideal[i]=0 ;}
                        else {
                                do {
                                                var_rand = ((double)
rand()/RAND_MAX);
                                                var = ( -DNL_ + var_rand * (
DNL_ + DNL_));
                                                tab_ref_not_ideal[i] =
tab_ref_not_ideal[i-1] + 1 + var;
                                                difference =
tab_ref_not_ideal[i] - tab_ref_ideal[i];
                                                }
                                while (abs(difference) > INL_);
                                        }

                        }
                using namespace std;
        #ifdef DEBUG
              cout << "-- " << name() << " (Dac)" << endl
                   << "   Vs  = " << Vs_ << " V" << endl
                   << "   Vsint1  = " << Vsint1_ << "V" << endl
                             << "   DNL = " << DNL_ << "Lsb" << endl
                             << "   INL = " << INL_ << "Lsb" << endl;
```

```
        #endif
        }
        void processing(){

                if ((sc_int<NBITS> (DigCont.read()) > 0)){

                        if (ABE_DAC_SENT_MODE == true)
                                out.write(tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())]*(Vsint1_/pow(2.0,NBITS)));

                         else
                                out.write(tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())]*(Vs.read()/pow(2.0,NBITS+1)));

                         current_dnl = tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())] -
                                                        (
tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())-1] +1);}

                else if ((sc_int<NBITS> (DigCont.read()) <0)){
                        if (ABE_DAC_SENT_MODE == true)
                                out.write(tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())+(int)pow(2.0,NBITS)]*(Vsint1_/pow(2.0,NBITS)));

                        else
                                out.write(tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())+(int)pow(2.0,NBITS)]*(Vs.read()/pow(2.0,NBITS+1)));
                        current_dnl = tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())+(int)pow(2.0,NBITS)] -
                                                        (
tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())+(int)pow(2.0,NBITS)-1] +1);}
                else current_dnl = 0.0;
                using namespace std;
                cout << current_dnl <<endl;

        }
private:
        double Vsint1_; // Internal Supply Voltage 1 [V]
        double DNL_;
        double INL_;
        std::vector<double> tab_ref_ideal; // table containing the ideal output
values in the absence of INL and DNL
        std::vector<double> tab_ref_not_ideal; // table containing the output
values if we consider the INL and DNL

};
#endif /* SRC_DAC_DNL_INL_TOP_DOWN_H_ */
```

Annex 18: **Testbench to simulate the module describing the effect of the INL and DNL on the DAC operation in a top down approach and valid for all possible input combination**

```
#include "Amplification.h"
#include "Dac_DNL_INL_TOP_DOWN.h"
#include "Sinesrc.h"
#include "Successive_generation_NBITSsequences.h"

int sc_main(int argc, char* argv[]) {
        using namespace sc_core;
       using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;
```

```
        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters
    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
        const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
    sc_clock clk_DAC("clk_DAC",CLK_PER, 0.5, SC_ZERO_TIME, false);
        const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals
        sca_tdf::sca_signal<double> Vout;
        sca_tdf::sca_signal<double> Voutampl;
        sca_tdf::sca_signal<double> vsrc;
        sc_signal<sc_bv<NBITS> > DigCont;
        sc_signal<bool> ABE_DAC_SENT_MODE;

        // Dac components

        ABE_DAC_SENT_MODE.write(true);

        Sinesrc i_ref("i_ref", AMPL, FREQ, OFFS);
        i_ref.out(vsrc);
        i_ref.set_timestep(TSTEP);
         inputcombination<NBITS> i_comb("i_comb");
         i_comb.clk(clk);
         i_comb.output(DigCont);
        Dac_DNL_INL<NBITS> i_dac("i_dac",VSINT1);
        i_dac.Vs(vsrc);
        i_dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
        i_dac.DigCont(DigCont);
        i_dac.out(Vout);
        i_dac.set_timestep(TSTEP);
        Amplification i_amp ("i_amp",GAIN);
        i_amp.in(Vout);
        i_amp.out(Voutampl);
        i_amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_DNL_INL_TOP_DOWN_tb");
         sca_util::sca_trace(tfp, DigCont, "DigCont");
     sca_util::sca_trace(tfp, Vout, "Vout");
         sca_util::sca_trace(tfp, Voutampl, "Voutampl");
         sca_util::sca_trace(tfp,ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
         sca_util::sca_trace(tfp,vsrc , "vsrc");
         sca_util::sca_trace(tfp,i_dac.current_dnl , "dnl");

         // simulation
         sc_start(TSTOP);
         return 0;
}
```

Annex 19: The NON-IDEAL DAC in a BOTTOM UP approach considering the temperature variation as well as the INL and DNL effect

```
#ifndef SRC_DAC_NOT_IDEAL_BOTTOM_UP_H_
#define SRC_DAC_NOT_IDEAL_BOTTOM_UP_H_

#include "systemc-ams.h"
#include <cstdlib> // for std::rand
#include <time.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <vector>
template <int NBITS> // NBITS digital control bits number

class Dac_DNL_INL_temp: public sca_tdf::sca_module {
public:

    sca_tdf::sca_de::sca_in<sc_bv<NBITS> > DigCont; // port input digital control
    sca_tdf::sca_de::sca_in<bool> ABE_DAC_SENT_MODE ; // ABE_DAC_SENT_MODE Signal
        sca_tdf::sca_out<double> out; //port output
        sca_tdf::sca_in<double> Vs; // supply voltage signal
        sca_tdf::sca_trace_variable<double> dnl_variation;
        sca_tdf::sca_trace_variable<double> inl_variation;

        void set_Temp(double T)
                { temp = T;}

        Dac_DNL_INL_temp(const sc_core::sc_module_name& name,
                                    double Vsint1 = 2.5,
                                    double T_amb = 37.0,
                                    double max_var = 0.37,
                                    double min_var = -0.37)
                            :  Vsint1_(Vsint1),T_amb_(T_amb), max_var_(max_var),
                                                min_var_(min_var),
DNL_(0.0),INL_(0.0),previous_output(0.0),
                                                weight_bit(NBITS),tab_ref_ideal((int)
pow(2.0,NBITS))

        {}
        virtual ~Dac_DNL_INL_temp() {}

protected:

        void initialize() {

                // function to guarantee the generation of random values
                srand(999);
                double var_rand; // random value between 0 and 1
                double var =0.0;

                for (int j = 0 ; j < NBITS; j++) {
                        var_rand = ((double) rand()/RAND_MAX);
                        var = min_var_ + var_rand* (max_var_ - min_var_);
                        weight_bit[j] = pow(2.0,j) + var ;}

                for (int i = 0 ; i < (int)(pow(2.0,NBITS)); i++) {

                    sc_bv<NBITS> x;
                        x = (sc_uint<NBITS>)(i); // coder l'entier i sur NBITS

                        double sum_ideal = 0.0;
                        for (int j = 0 ; j < NBITS; j++) {
                                if (x[j] == 1 ) {
                                                sum_ideal += pow(2.0,j) ;}
                                }
                        tab_ref_ideal[i]=sum_ideal;}
            using namespace std;
#ifdef DEBUG


      cout << "-- " << name() << " (Dac)" << endl
           << "   Vs  = " << Vs_ << " V" << endl
           << "   Vsint1  = " << Vsint1_ << "V" << endl
                << "   DNL = " << DNL_ << "Lsb" << endl
                << "   INL = " << INL_ << "Lsb" << endl;
```

```
#endif

}
        void processing(){

                double output = 0.0;
                double inter = 0.0 ;
                double sum = 0.0 ;
                for (int i = 0 ; i < NBITS; i++) {
                        if ((sc_uint<NBITS>)DigCont.read().get_bit(i) == 1 ) {
                        sum += weight_bit[i] ;}
                                }

                output = sum ;
                if (ABE_DAC_SENT_MODE == true)
                        inter = output*(Vsint1_/pow(2.0,NBITS));

                else
                        inter = output*(Vs.read()/pow(2.0,NBITS+1));

                if (temp == T_amb_)
                        out.write(inter);
                else
                        out.write (inter-inter*(temp - T_amb_)/ T_amb_);

                if (sc_int<NBITS> (DigCont.read()) < 0){
                        dnl_variation = output - (1 + previous_output);
                        inl_variation = output - tab_ref_ideal[(sc_int<NBITS>)
(DigCont.read()) + (int) pow(2.0,NBITS)];}

                else if (sc_int<NBITS> (DigCont.read()) > 0)
                         {dnl_variation = output - (1 + previous_output);
                          inl_variation = output -
tab_ref_ideal[(sc_int<NBITS>)(DigCont.read())];}

                else if (sc_int<NBITS> (DigCont.read()) == 0)
                        {dnl_variation = 0 ;
                         inl_variation =0 ;}

                if (abs (DNL_)< abs (dnl_variation)) DNL_ = dnl_variation;
                if (abs (INL_) < abs (inl_variation)) INL_ = inl_variation;


                        using namespace std;
                                cout << "-- " << "DNL" << DNL_ << endl
                                     << "-- " << "INL" << INL_ << endl;
                previous_output=output;

        }
private:
        double Vsint1_; // Internal Supply Voltage 1 [V]
        double T_amb_;  // temperature ambiante
        double max_var_; // max bit weight variation
        double min_var_; // min bit weight variation
        double DNL_;
        double INL_;
        double previous_output;
        std::vector<double> weight_bit; // table containing random bit weight
        std::vector<double> tab_ref_ideal; // table containing ideal output values
        double temp;

};

#endif /* SRC_DAC_DNL_INL_H_ */
```

Annex 20: **Temperature variation module considering the INL and DNL in the BOTTOM UP approach**

```
#ifndef SRC_TEMP_VAR_DNL_INL_H_
#define SRC_TEMP_VAR_DNL_INL_H_
#include "Successive_generation_NBITSsequences.h"
#include "Amplification.h"
#include "Dac_NOT_IDEAL_BOTTOM_UP.h"
#include "Sinesrc.h"
#include "systemc-ams.h"
#include <cstdlib> // for std::rand
template <int NBITS> // NBITS digital control bits number

class tempvariation: public sc_module {
public:

        sc_in<bool> clk;
        sc_in<bool> clk_temp;
        sca_tdf::sca_signal<double> Vout;
        sca_tdf::sca_out<double> Voutampl;
        sca_tdf::sca_signal<double> vsrc;
        sc_signal<bool> ABE_DAC_SENT_MODE;;
        sc_signal<sc_bv<NBITS> > DigCont;


        Sinesrc Sin;
        Dac_DNL_INL_temp<NBITS> Dac;
        inputcombination<NBITS> digital;
        Amplification amp;

         void temperature_update () {

                double T;
                double x;
                while (true)
                 {
                x =  ((double)std::rand())/RAND_MAX ;
                T = T_MIN_ + x * (T_MAX_-T_MIN_);
                Dac.set_Temp(T);
                wait();
                using namespace std;
                cout << "-- " << T << " T " << endl;
                 }
           }
         SC_HAS_PROCESS(tempvariation);

        tempvariation(const sc_core::sc_module_name& name,
                                double VSINT1 = 2.5 , // [V]
                            double GAIN = 2.0,
                            double T_AMB = 37.0,
                            double AMPL = 1.0,
                            double OFFS = 0.0,
                            double FREQ = 1.0e6,
                            double T_MAX = 40.0,
                            double T_MIN = 35.0,
                                double max_var = 0.37,
                                double min_var = -0.37
                                )

          :  VSINT1_(VSINT1), GAIN_(GAIN), T_AMB_(T_AMB),AMPL_(AMPL),

OFFS_(OFFS),FREQ_(FREQ),T_MAX_(T_MAX),T_MIN_(T_MIN),max_var_(max_var),min_var_(min_
var),
                clk ("clk"), Vout("Vout"), Voutampl("Voutampl"),vsrc("vsrc"),
                DigCont("DigCont"),ABE_DAC_SENT_MODE("ABE_DAC_SENT_MODE"),
```

```
                  Sin("Sin",AMPL, FREQ,
OFFS),Dac("Dac",VSINT1,T_AMB,max_var,min_var),digital("digital"),
                  amp("amp")

        {
                 ABE_DAC_SENT_MODE.write(true);
        // sin component
                 Sin.out(vsrc);
        // Digital control component
                 digital.clk(clk);
                 digital.output(DigCont);
        // DAC component
                Dac.Vs(vsrc);
                Dac.ABE_DAC_SENT_MODE(ABE_DAC_SENT_MODE);
                Dac.DigCont(DigCont);
                Dac.out(Vout);
        // amplification component
                amp.in(Vout);
                amp.out(Voutampl);

                SC_THREAD(temperature_update);
                sensitive << clk.pos();
}

        virtual ~tempvariation() {}

protected:
private:
             double VSINT1_ ; // [V]
             double GAIN_ ;
             double T_AMB_ ;
             double AMPL_ ;
             double OFFS_ ;
             double FREQ_ ;
             double T_MAX_ ;
             double T_MIN_ ;
             double DNL_;
             double INL_;
             double max_var_;
             double min_var_;
};

#endif /* SRC_DAC_TEMP_H_ */
```

Annex 21: **Testbench to observe the INL and DNL effect on the NON-IDEL DAC in the BOTTOM UP approach**

```
#include "Amplification.h"
#include "Dac_NOT_IDEAL_BOTTOM_UP.h"
#include "Sinesrc.h"
#include "systemc-ams.h"
#include <cstdlib> // for std::rand

#include "temp_var_DNL_INL.h"
int sc_main(int argc, char* argv[]) {

        using namespace sc_core;
        using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;
        const double T_AMB = 37.0;
        const double T_MAX = 37.0;
        const double T_MIN = 37.0;
```

```
        const double MAX_VAR= 0.37;
        const double MIN_VAR = -0.37;

        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters

    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
        const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
    sc_clock clk_temp("clk_temp", CLK_PER, 0.5, SC_ZERO_TIME, false);
        const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals
        sca_tdf::sca_signal<double> Voutampl;

        tempvariation<NBITS> i_test
("i_test",VSINT1,GAIN,T_AMB,AMPL,OFFS,FREQ,T_MAX,T_MIN,MAX_VAR,MIN_VAR);
        i_test.clk_temp(clk_temp);
        i_test.clk(clk);
        i_test.Voutampl(Voutampl);
        i_test.Dac.set_timestep(TSTEP);
        i_test.Sin.set_timestep(TSTEP);
        i_test.amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_temp_INL_DNL_SENTMODE_tb");
         sca_util::sca_trace(tfp, i_test.DigCont, "DigCont");
     sca_util::sca_trace(tfp, i_test.Vout, "Vout");
         sca_util::sca_trace(tfp, Voutampl, "Voutampl");
         sca_util::sca_trace(tfp,i_test.ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
         sca_util::sca_trace(tfp,i_test.vsrc , "vsrc");
         sca_util::sca_trace(tfp,i_test.Dac.dnl_variation , "dnl_variation");
         sca_util::sca_trace(tfp,i_test.Dac.inl_variation , "inl_variation");

         // simulation
         sc_start(TSTOP);
         return 0;
}
```

Annex 22: **Testbench to observe the temperature variation effect on the NON-IDEL DAC in the BOTTOM UP approach**

```
#include "Amplification.h"
#include "Dac_NOT_IDEAL_BOTTOM_UP.h"
#include "Sinesrc.h"
#include "systemc-ams.h"
#include <cstdlib> // for std::rand

#include "temp_var_DNL_INL.h"
int sc_main(int argc, char* argv[]) {

        using namespace sc_core;
        using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;
        const double T_AMB = 37.0;
```

```
        const double T_MAX = 40.0;
        const double T_MIN = 35.0;
        const double MAX_VAR= 0.0;
        const double MIN_VAR = 0.0;

        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters

    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
        const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
    sc_clock clk_temp("clk_temp", CLK_PER, 0.5, SC_ZERO_TIME, false);
        const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals

        sca_tdf::sca_signal<double> Voutampl;

        tempvariation<NBITS> i_test
("i_test",VSINT1,GAIN,T_AMB,AMPL,OFFS,FREQ,T_MAX,T_MIN,MAX_VAR,MIN_VAR);
        i_test.clk_temp(clk_temp);
        i_test.clk(clk);
        i_test.Voutampl(Voutampl);
        i_test.Dac.set_timestep(TSTEP);
        i_test.Sin.set_timestep(TSTEP);
        i_test.amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_temp_INL_DNL_SENTMODE_tb");
         sca_util::sca_trace(tfp, i_test.DigCont, "DigCont");
     sca_util::sca_trace(tfp, i_test.Vout, "Vout");
         sca_util::sca_trace(tfp, Voutampl, "Voutampl");
         sca_util::sca_trace(tfp,i_test.ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
         sca_util::sca_trace(tfp,i_test.vsrc , "vsrc");
         sca_util::sca_trace(tfp,i_test.Dac.dnl_variation , "dnl_variation");
         sca_util::sca_trace(tfp,i_test.Dac.inl_variation , "inl_variation");

         // simulation
         sc_start(TSTOP);
         return 0;
}
```

Annex 23: **The NON-IDEAL DAC in a TOP DOWN approach considering the temperature variation as well as the INL and DNL effect**

```
#ifndef SRC_DAC_NOT_IDEAL_TOP_DOWN_H_
#define SRC_DAC_NOT_IDEAL_TOP_DOWN_H_
#include "systemc-ams.h"
#include <cstdlib> // for std::rand
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <vector
template <int NBITS> // NBITS digital control bits number

class Dac_DNL_INL_temp: public sca_tdf::sca_module {
public:

    sca_tdf::sca_de::sca_in<sc_bv<NBITS> > DigCont; // port input digital control
"12 bits signal"
    sca_tdf::sca_de::sca_in<bool> ABE_DAC_SENT_MODE ; // ABE_DAC_SENT_MODE Signal
```

```
        sca_tdf::sca_out<double> out; //port output
        sca_tdf::sca_in<double> Vs; // supply voltage signal
        sca_tdf::sca_trace_variable<double> current_dnl;

        void set_Temp(double T)
        { temp = T;}

        Dac_DNL_INL_temp(const sc_core::sc_module_name& name,
                                    double Vsint1 = 2.5,
                                    double DNL = 1.5,
                                    double INL = 2.0,
                                    double T_amb = 35.0)
                            :  Vsint1_(Vsint1),
DNL_(DNL),INL_(INL),T_amb_(T_amb),tab_ref_ideal((int)pow(2.0,NBITS)),
                                            tab_ref_not_ideal((int)pow(2.0,NBITS))


        {}
        virtual ~Dac_DNL_INL_temp() {}

protected:
        void initialize() {
                // function to guarantee the generation of random values
                srand(999);
                for (int i = 0 ; i < (int)(pow(2.0,NBITS)) ; i++) {

                    double var_rand; // random value between 0 and 1
                    double var;           //      random value between DNL and -
DNL
                    double difference = 0.0; // difference between ideal value and
not ideal value to conclude the INL
                    sc_bv<NBITS> x;
                        x = (sc_uint<NBITS>)(i); // coder l'entier i sur NBITS

                        // Reference table of ideal values

                        double sum_ideal = 0.0;
                        for (int j = 0 ; j < NBITS; j++) {
                                if (x[j] == 1 ) {
                                                sum_ideal += pow(2.0,j) ;}
                                }
                        tab_ref_ideal[i]=sum_ideal;

                        // Reference table of not ideal values respecting DNL and
INL

                        if (i==0) { tab_ref_not_ideal[i]=0 ;}
                        else {
                                do {
                                                var_rand = ((double)
rand()/RAND_MAX);
                                                var = ( -DNL_ + var_rand * (
DNL_ + DNL_));
                                                tab_ref_not_ideal[i] =
tab_ref_not_ideal[i-1] + 1 + var;
                                                difference =
tab_ref_not_ideal[i] - tab_ref_ideal[i];
                                                }
                                while (abs(difference) > INL_);
                                        }

                        }
                using namespace std;
        #ifdef DEBUG
```

```
                cout << "-- " << name() << " (Dac)" << endl
                     << "   Vs   = " << Vs_ << " V" << endl
                     << "   Vsint1   = " << Vsint1_ << "V" << endl
                               << "   DNL = " << DNL_ << "Lsb" << endl
                               << "   INL = " << INL_ << "Lsb" << endl;

        #endif
        }
        void processing(){
                double inter;
                if ((sc_int<NBITS> (DigCont.read()) > 0)){

                           if (ABE_DAC_SENT_MODE == true)
                                   inter = (tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())]*(Vsint1_/pow(2.0,NBITS)));

                            else
                                   inter = (tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())]*(Vs.read()/pow(2.0,NBITS+1)));

                            current_dnl = tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())] -
                                                                (
tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())-1] +1);}

                else if ((sc_int<NBITS> (DigCont.read()) <0)){
                           if (ABE_DAC_SENT_MODE == true)
                                   inter = (tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())+(int)pow(2.0,NBITS)]*(Vsint1_/pow(2.0,NBITS)));

                           else
                                   inter = (tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())+(int)pow(2.0,NBITS)]*(Vs.read()/pow(2.0,NBITS+1)));
                           current_dnl = tab_ref_not_ideal[sc_int<NBITS>
(DigCont.read())+(int)pow(2.0,NBITS)] -
                                                                (
tab_ref_not_ideal[sc_int<NBITS> (DigCont.read())+(int)pow(2.0,NBITS)-1] +1);}
                else current_dnl = 0.0;

                if (temp == T_amb_)
                        out.write(inter);
                else
                        out.write (inter-inter*(temp - T_amb_)/ T_amb_);
                using namespace std;
                cout << current_dnl <<endl;


private:
        double Vsint1_; // Internal Supply Voltage 1 [V]
        double DNL_;
        double INL_;
        double T_amb_; // temperature ambiante
        std::vector<double> tab_ref_ideal; // table containing the ideal output
values in the absence of INL and DNL
        std::vector<double> tab_ref_not_ideal; // table containing the output
values if we consider the INL and DNL
        double temp;

};
#endif /* SRC_DAC_NOT_IDEAL_TOP_DOWN_H_ */
```

Annex 24: **Testbench to observe the INL and DNL effect on the NON-IDEL DAC in the TOP DOWN approach**

```
#include "Amplification.h"
#include "DAC_NOT_IDEAL_TOP_DOWN.h"
#include "Sinesrc.h"
```

```
#include "systemc-ams.h"
#include <cstdlib> // for std::rand

#include "temp_var_DNL_INL.h"
int sc_main(int argc, char* argv[]) {

        using namespace sc_core;
       using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;
        const double T_AMB = 37.0;
        const double T_MAX = 37.0;
        const double T_MIN = 37.0;
        const double DNL = 1.5;
        const double INL = 2.0;

        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters

    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
        const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
    sc_clock clk_temp("clk_temp", CLK_PER, 0.5, SC_ZERO_TIME, false);
        const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals
        sca_tdf::sca_signal<double> Voutampl;

        tempvariation<NBITS> i_test
("i_test",VSINT1,GAIN,T_AMB,AMPL,OFFS,FREQ,T_MAX,T_MIN,DNL,INL);
        i_test.clk_temp(clk_temp);
        i_test.clk(clk);
        i_test.Voutampl(Voutampl);
        i_test.Dac.set_timestep(TSTEP);
        i_test.Sin.set_timestep(TSTEP);
        i_test.amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_temp_INL_DNL_SENTMODE_tb");
         sca_util::sca_trace(tfp, i_test.DigCont, "DigCont");
     sca_util::sca_trace(tfp, i_test.Vout, "Vout");
     sca_util::sca_trace(tfp, i_test.Dac.current_dnl, "current_dnl");
         sca_util::sca_trace(tfp, Voutampl, "Voutampl");
         sca_util::sca_trace(tfp,i_test.ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
         sca_util::sca_trace(tfp,i_test.vsrc , "vsrc");

         // simulation
         sc_start(TSTOP);
         return 0;
}
```

Annex 25: **Testbench to observe the temperature effect on the NON-IDEL DAC in the TOP DOWN approach**

```
#include "Amplification.h"
#include "DAC_NOT_IDEAL_TOP_DOWN.h"
```

```
#include "Sinesrc.h"
#include "systemc-ams.h"
#include <cstdlib> // for std::rand

#include "temp_var_DNL_INL.h"

int sc_main(int argc, char* argv[]) {

        using namespace sc_core;
    using namespace sca_core;
        using namespace sca_util;

        // system parameters
        const int NBITS = 12;
        const double VSINT1 = 2.5; // [V]
        const double GAIN = 2.0;
        const double T_AMB = 37.0;
        const double T_MAX = 40.0;
        const double T_MIN = 35.0;
        const double DNL = 0.0;
        const double INL = 0.0;

        // source parameters
        const double AMPL = 1.0;
        const double OFFS = 0;
        const double FREQ = 1.0e6;

        // simulation parameters

    const double NTPTS_PER_PERIOD = 50;  // number of timepoints per period
    const sca_time TSTEP = sca_time(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
        const sc_time CLK_PER(1/FREQ/NTPTS_PER_PERIOD, SC_SEC);
    sc_clock clk("clk", CLK_PER, 0.5, SC_ZERO_TIME, false);
    sc_clock clk_temp("clk_temp", CLK_PER, 0.5, SC_ZERO_TIME, false);
        const sca_time TSTOP = (int)pow(2.0, double(NBITS))*CLK_PER;

        // signals

        sca_tdf::sca_signal<double> Voutampl;

        tempvariation<NBITS> i_test
("i_test",VSINT1,GAIN,T_AMB,AMPL,OFFS,FREQ,T_MAX,T_MIN,DNL,INL);
        i_test.clk_temp(clk_temp);
        i_test.clk(clk);
        i_test.Voutampl(Voutampl);
        i_test.Dac.set_timestep(TSTEP);
        i_test.Sin.set_timestep(TSTEP);
        i_test.amp.set_timestep(TSTEP);

         // tracing
         sca_util::sca_trace_file *tfp =
sca_util::sca_create_vcd_trace_file("dac_temp_INL_DNL_SENTMODE_tb");
         sca_util::sca_trace(tfp, i_test.DigCont, "DigCont");
     sca_util::sca_trace(tfp, i_test.Vout, "Vout");
     sca_util::sca_trace(tfp, i_test.Dac.current_dnl, "current_dnl");
         sca_util::sca_trace(tfp, Voutampl, "Voutampl");
         sca_util::sca_trace(tfp,i_test.ABE_DAC_SENT_MODE , "ABE_DAC_SENT_MODE");
         sca_util::sca_trace(tfp,i_test.vsrc , "vsrc");

         // simulation
         sc_start(TSTOP);
         return 0;
}
```

Annex 26: **CADENCE environment set-up instructions**

•Setup Xcelium path and gcc path:

```
setenv PATH /path/to/Xcelium/install/tools/bin:$PATH
setenv PATH `xmroot`/tools/cdsgcc/gcc/bin:$PATH
• To be able to load the needed shared libraries on compilation and execution, the
environment variable LD_LIBRARY_PATH needs to be set correctly. Paths it must
contain:
o Xcelium's SystemC library path
o Library path for Xcelium's GCC
o SystemC-AMS library path (for using the library; see prefix from configure step)
setenv LD_LIBRARY_PATH `xmroot`/tools/systemc/lib/64bit:$LD_LIBRARY_PATH
setenv LD_LIBRARY_PATH `xmroot`/tools/cdsgcc/gcc/6.3/install/lib64:$LD_LIBRARY_PATH
•Option 1: create a libsystemc-ams.a, with no change in the build system
1. Unpack the tarball systemc-ams-2.3.tar.gz
2. (Optionally) patch the sources to get instrumentation in top level folder
$> patch -p0 < ./scams_instrumentation.patch
3. Create a build directory
$> mkdir objdir && cd objdir
4. Configure the build system
$> ../configure \
--with-systemc=`xmroot`/tools/systemc \
CXXFLAGS="-fPIC -O3 -g -Wall -DNCSC \
-DSYSTEMC_VERSION=20130101 \
-D_GLIBCXX_USE_CXX11_ABI=0" \
CPPFLAGS="-fPIC -O3 -g -Wall -DNCSC \
-DSYSTEMC_VERSION=20130101 \
-D_GLIBCXX_USE_CXX11_ABI=0" \
--prefix=/path/to/installation \
--disable-systemc_compile_check \
--disable-shared
5. make
6. make install
•Option 2: also create a libsystemc-ams.so, with a patch on the build system
1. Unpack the tarball systemc-ams-2.3.tar.gz
2. Patch the build system
$> patch -p0 < ./scams_xcelium.patch
3. (Optionally) patch the sources to get instrumentation in top level folder
$> patch -p0 < ./scams_instrumentation.patch
4. Create a build directory
$> mkdir objdir && cd objdir
5. Configure the build system
$> ../configure \
--with-systemc=`xmroot`/tools/systemc \
CXXFLAGS="-fPIC -O3 -g -Wall -DNCSC \
-DSYSTEMC_VERSION=20130101 \
-D_GLIBCXX_USE_CXX11_ABI=0" \
CPPFLAGS="-fPIC -O3 -g -Wall -DNCSC \
-DSYSTEMC_VERSION=20130101 \
-D_GLIBCXX_USE_CXX11_ABI=0" \
--prefix=/path/to/installation \
--disable-systemc_compile_check
6. make
7. make install
To run it with libsystemc-ams.so, we use xrun
$> xrun -64bit -sysc \
-I$SYSTEMC_AMS_HOME/include \
-L$SYSTEMC_AMS_HOME/lib-linux64 -lsystemc-ams \
basic_mixer_tdf_de.cpp \
-gui
• To run it with libsystemc-ams.a, we use xrun with a different lib option
$> xrun -64bit -sysc \
-I$SYSTEMC_AMS_HOME/include \
$SYSTEMC_AMS_HOME/lib-linux64/libsystemc-ams.a \
basic_mixer_tdf_de.cpp \
-gu
```

Abstract

As a part of my master thesis project, I did my 6-month internship in Melexis, a Semiconducting company in Bevaix, Switzerland. I was part of the mixed-signal ICs design team.

There are two main parts in this project. The first one is to develop some knowledge on the SystemC AMS which is an extension of the SystemC and essentially introduces the design and modeling of integrated analog and mixed-signal systems. The second part is devoted to the development of models describing the ideal operation of the DAC of a Triaxis Analog Backend and the study of some non-idealities such as temperature fluctuations and the effects of INL and DNL, then verifying these models in a simple environment using open-source tools (Eclipse) and using the Cadence tool environment available in Melexis to explore the capabilities of SystemC AMS.

Due to the exceptional health and epidemic context, the work was carried out remotely by teleworking.

Résumé

Dans le cadre de mon projet de fin d'études, j'ai effectué mon stage de 6 mois dans l'entreprise de Semiconducting Melexis et qui se situe à Bevaix en Suisse. Je faisais partie de l'équipe de conception des circuits intégrés à signaux mixtes.

Ce projet comporte deux grandes parties. La première consiste à développer certaines connaissances sur le SystemC AMS qui est une extension du SystemC et qui introduit essentiellement la conception et la modélisation des systèmes à signaux analogiques et mixtes intégrés. La deuxième partie est consacrée au développement des modèles décrivant le fonctionnement idéal du DAC du Triaxis Analog Backend et l'étude de certaines non-idéalités telles que les fluctuations de température et les effets de l'INL et du DNL, ensuite vérifier ces modèles développés dans un environnement simple à l'aide d'outils open-source (Eclipse) et en utilisant l'environnement de l'outil Cadence disponible dans Melexis pour explorer les capacités du SystemC AMS.

A cause du contexte sanitaire exceptionnel, le travail a été réalisé à distance en télétravail.

Abstratto

Nella mia tesi di master, ho svolto il mio stage di 6 mesi in Melexis, un'azienda di semiconduttori situata in Bevaix, Svizzera. Ho fatto parte del team di design di circuiti integrati con segnali misti.

Questo progetto è costituito da due parti principali. La prima parte consiste nello sviluppare le conoscenze necessarie su SystemC AMS, un estensione di SystemC, ed introduce essenzialmente il design e il modellamento di sistemi integrati analogici e a segnali misti. La seconda parte è dedicata allo sviluppo di modelli che descrivano il comportamento ideale del DAC all' interndo di un Triaxis Analog e allo studio di alcune non-idealità, come ad esempio fluttuazioni in temperatura e gli effetti di non-linearità integrale (INL) e differenziale (DNL), verificando poi questi modelli in un ambiente semplice utilizzando dei software open-source (Eclipse) e utilizzando la versione di cadence fornita da Melexis per esplorare il potenziale di SistemC AMS.

A causa dello stato di epidemia e di emergenza sanitaria, questo lavoro è stato svolto remotamente in maniera telematica.