Analog Transaction Level Modeling for Verification of Mixed-Signal-Blocks

Alexander W. Rath*, Volkan Esen† and Wolfgang Ecker*
*Infineon Technologies AG
85579 Neubiberg, Germany
Technische Universität München
Email: *Firstname.Lastname*@infineon.com
†Infineon Technologies AG
Email: Volkan.Esen@infineon.com

***Abstract*—The Universal Verification Methodology (UVM) has become a de facto standard in today's functional verification of digital designs. However, it is rarely used for the verification of mixed-signal designs. This paper presents a new abstraction technique using UVM that can be used in order to stimulate mixed signal designs. It is referred to as Analog Transaction Level Modeling.**

I. Introduction

In today's IC designs more and more parts of the analog implementation are shifted to the digital domain, since digital circuits scale better with new technologies. This trend leads to mixed signals designs. Their analog and digital parts interface with each other as well as with the outside world (see fig. 1).

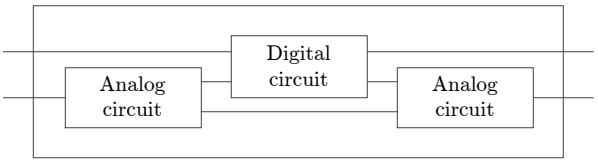

Fig. 1. Structure of a mixed signal design containing analog and digital circuits interfacing with each other and the outside world

The functional verification of the analog parts is different compared to the verification of the digital parts:

- Digital parts are functionally verified on register transfer level (RTL) using very sophisticated methodologies like OVM [1] or UVM [2]. Their key concepts are the generation of constrained-random stimulus, automated checking mechanisms and the collection of functional coverage.
- The analog parts of mixed signal designs are usually verified on SPICE level using analog network simulators. This approach covers mainly the verification of electrical parameters, e. g. input resistance and amplification. However, it is also used to verify the functional behavior of the block.

However, in the verification of the whole chip (chip level verification), where the system-level behavior as well as the interconnectivity of the blocks are to be checked, the detailedness of the SPICE models is often not required. Also, they slow down the simulation speed drastically.

In consequence, it is a common practice not to use the SPICE models in chip level simulations. Instead, so called real number models (RNM) are used. They purely reflect the functional behavior of the analog parts and are developed by using a hardware description language, e. g. VHDL, Verilog or SystemVerilog. The advantage of this approach is that a regular event driven simulator can be used to perform the chip level verification.

However, the resulting design under test (DUT) has not only digital inputs and outputs. Some of them are analog instead, as shown in fig. 1. The degree of freedom for analog – real-valued – signals is much higher as it is for usual digital signals, i. e. their co-domain is nearly unlimited.

Today, the stimulus for the real-valued inputs is usually done in a directed way, i. e. it is hard coded in the tests. Randomization is rarely used.

Hence, in this paper, we present, how the constraint random stimulation of such a DUT can be efficiently achieved by introducing the concept of analog transactions.

The paper is structured as follows. First, we give an overview about related work in this field. Following that, we show how the concept of transactions is used in UVM testbenches to stimulate digital designs. In connection to that we present how we extent this concept to analog stimulus. Finally, we show two example applications.

II. Related Work

UVM is the emerging de facto standard for creating reusable testbenches and verification environments. Released by Accellera, this standard defines a class library, which allows verification engineers to build verification components (VCs) and environments in a standardized way. Further, the UVM class library provides a callback mechanism, which enables VCs and system models to communicate via TLM.

For the analog domain no such abstract communication technique is available. However, several different approaches to extend modern hardware, verification and system description languages with the ability to describe analog behavior have been developed; the newest being SystemC-AMS, presented in [3]. SystemC-AMS allows modeling engineers to describe analog behavior in frequency and time domain.

The drawback is, that no verification library, that is such sophisticated as UVM, exists for SystemC or SystemC-AMS.

Another new approach is UVM-MS presented in [4]. This approach focuses mainly on the direct stimulation of the pins of the DUT using UVM and an additional Verilog-AMS layer.

All these newer approaches for the analog domain enable the verification of AMS models. The difference between AMS models and the aforementioned RNMs is that AMS models aim more at a higher level of electrical accuracy that is often not required for chip level verification. In consequence, AMS techniques are not fitting well to the verification problem described in section I.

III. Stimulus generation using UVM

In this section, we shortly explain how stimulus generation for digital designs is done using UVM.

In order to stimulate a DUT using UVM, transactions are used. That means, that the input pins of a DUT are not driven directly from the test. Instead, a data structure containing parameters is passed to a driver. The parameters are reflecting the functional features of a certain protocol. For example for stimulating a serial interface, these parameters are the address and the payload. When issuing the transaction from the test, the parameters are set using constrained randomization. Once the driver receives the transaction, it drives the pins according to the protocol and according to the parameters delivered by the transaction. Hence, from a test point of view, transactions offer abstraction of the protocol.

To make use of this technique, the transaction's data structure has to be defined and a driver has to be implemented. To implement the data structure, the predefined class `uvm_sequence_item` is extended. See listing 1 for an example.

```
1  class serial_interface_seq_item extends \
2  uvm_sequence_item;
3   rand bit[ 7:0] addr;
4   rand bit[15:0] payload;
5
6   `uvm_object_utils_begin \
7   (serial_interface_seq_item)
8    `uvm_field_int(addr,    UVM_ALL_ON)
9    `uvm_field_int(payload, UVM_ALL_ON)
10  `uvm_object_utils_end
11 endclass
```

Listing 1. Example of a sequence item for a serial interface containing the parameters `addr` and `payload`. The keyword `rand` enables the randomization when issuing the transaction.

In order to implement the driver, the `uvm_driver` class is to be extended. In the driver's run phase, the received instances of the class `serial_interface_seq_item` are decoded and the pins of DUT are driven as defined by the protocol's standard.

IV. Analog stimulus generation

In the following section we show, how the concept of transactions as shown in the previous section can be extended to analog stimulus.

Analog signals are different comparing to digital signals, as their co-domain is nearly unlimited. That allows analog signals to adopt different shapes, whereas a single digital signal always has a rectangular shape. However, it is possible to classify the shape of an analog signal. For example an analog signal can be of a linear, harmonic or cubic spline shape or of any other shape as well. To describe an analog signal in detail, it is not sufficient to simply name the shape. Additional parameters are required. For example, to describe a linear signal the slope as well as one value at a certain point in time are to be known.

In the abstraction approach presented in this paper, we identify the term "shape" with the term "protocol" from the previous section. That means that we use a `uvm_driver` to generate an analog signal of a certain shape with parameters delivered by a randomized `uvm_sequence_item`.

In the following subsection we show how harmonic and cubic spline shaped stimulus can be generated with this technique. Stimulus for signals shaped differently could be generated using the same technique.

A. Fourier *transformation-based stimulus*

1) Idea: In many applications harmonic analog signals play a big role. This is especially true for filters and amplifiers. Harmonic signals can be described by their spectrum. Hence, the spectrum is the parameter set describing the signal. From a spectrum the harmonic signal can be gained through an inverse Fourier transformation. In consequence, a `uvm_sequence_item` for harmonic signals carries a randomized spectrum of the signal to be generated and the according `uvm_driver` has to perform an inverse Fourier transformation.

2) Basic theory: In this subsection, we present the basic theory for the Fourier transformation referring to [5].

Harmonic signals can be written like this:

$$s(t) = \sum_{n=-\infty}^{\infty} c_n \mathrm{e}^{\mathrm{i}n\omega_0 t}, \tag{1}$$

where $s(t)$ is the signal over time, i the imaginary unit, $\omega_0 = \frac{2\pi}{T_0}$ the angular fundamental frequency and T_0 the period of the signal $s(t)$. Equation 1 is called the Fourier series expansion of $s(t)$. For any sequence (c_n) the signal is determined bijectively. c_n are the Fourier coefficients of the signal. In general, the co-domain of c_n are the complex numbers.

The kernel $K_n(t) = \mathrm{e}^{\mathrm{i}n\omega t}$ of the Fourier series expansion of $s(t)$ is a periodic and also harmonic function, which means, that a particular kernel $K_j(t)$, multiplied with its respective Fourier coefficient c_j, represents one spectral portion of the signal $s(t)$. Therefore the sequence (c_n) is also called the spectrum of $s(t)$.

The process of calculation the respective spectrum from a signal $s(t)$ is called the FOURIER transformation, with an arbitrary point of time t_0:

$$S(\omega) = \frac{1}{T_0} \int_{t_0}^{t_0+T_0} s(t) \mathrm{e}^{-\mathrm{i}\omega t} \, \mathrm{d}t. \tag{2}$$

The function $S(\omega)$ is called the FOURIER transformed of $s(t)$. From this function, the spectrum can be obtained by setting

$$c_n = S(n\omega_0), \text{ with } n \in \mathbb{N}. \tag{3}$$

Often the transformation term in (2) is abbreviated with

$$S(\omega) = \mathcal{F}\left(s(t)\right). \tag{4}$$

A `uvm_driver` will have to calculate a signal $s(t)$ from the spectrum (c_n). This process is called the inverse FOURIER transformation, denoted with the symbol $\mathcal{F}^{-1}\left(S(\omega)\right)$. Equation (1) can be used to perform this process.

3) Implementation: To implement the inverse FOURIER transformation we used FFTW [6]. FFTW is written in C. Most modern languages used in digital design offer an interface to the C language, which allows verification engineers, to make use of FFTW also in their projects.

Based on FFTW, SystemVerilog and its Direct Programming Interface (DPI; [7]), we developed a library, that allows using it in UVM based testbenches. The core of this library consists of a C wrapper for FFTW, that is callable via the DPI. To hide the implementation details we developed a UVM compliant class library that alleviates the use of FFTW. The class library is implemented as a package which is shown in listing 2.

The package `fourier_pkg` imports another package `sv_complex_pkg`. This package defines a complex data type and is shown in listing 3. Furthermore, `fourier_pkg` defines DPI function prototypes. They serve as an interface between the C and the SystemVerilog world. On the SystemVerilog side, their arguments are dynamic arrays. The first one being a dynamic array containing data that are to be transformed. The second on being the transformed result. On the C side, their arguments are pointers on the respective data.

This approach of passing pointers through the interface allows the memory allocation being done on the SV side which has basically three advantages:

1) Using dynamic arrays on the SystemVerilog side allows typed memory allocation, whereas C only allows untyped memory allocation using `malloc()`.
2) The resulting data are provided in a normal dynamic array. There is no need of accessing the C interface in order to obtain transformed data.
3) SystemVerilog provides garbage collection. That means that the user has not to free any allocated memory.

To make the usage even simpler, we introduced wrapper classes that hides the DPI access. They are extending `uvm_object`. Therefore they integrate themselves seamlessly into a UVM test environment. Furthermore they allocate the dynamic array that will contain the result, i. e. the transformed data.

```
package fourier_pkg;
import sv_complex_pkg::*;

import uvm_pkg::*;
`include "uvm_macros.svh";

import "DPI-C" function void
  fourier_transformation_dpi
  (real to_be_transformed[],
   inout sv_complex transformed[]);

import "DPI-C" function void
  fourier_inverse_transformation_dpi
  (sv_complex to_be_transformed[],
   inout real transformed[]);

class fourier_transformation
  extends uvm_object;
  sv_complex result[];

  `uvm_object_utils
    (fourier_transformation)

  function void fourier_transformation
    (real to_be_transformed[]);
    result = new[...];
    fourier_transformation_dpi
     (to_be_transformed, result);
endclass: fourier_transformation

class fourier_inverse_transformation
  extends uvm_object;
  real result[];

  `uvm_object_utils
    (inverse_fourier_transformation)

  function void
    fourier_inverse_transformation
    (sv_complex to_be_transformed[]);
    result = new[...];
    fourier_inverse_transformation_dpi
      (to_be_transformed, result);
endclass: fourier_inverse_transformation
endpackage: fourier_pkg
```

Listing 2. Package for Fourier transformation containing the DPI function declarations and the wrapper classes.

Besides these core functions, we developed also a sequence item base class, modeling an analog transaction. It is passed to a `uvm_driver` that performs the inverse FOURIER transformation using the package described above and stimulates the design under test. The sequence item class contains a dynamic array on complex numbers, i. e. the spectrum and

```
package sv_complex_pkg;
typedef struct{real re; real im;}
  sv_complex;

function sv_complex sv_cadd(sv_complex c1,
  sv_complex c2);
  return '{c1.re + c2.re,
    c1.im + c2.im};
endfunction: sv_cadd

function sv_complex sv_csub(sv_complex c1,
  sv_complex c2);
  return '{c1.re - c2.re,
    c1.im - c2.im};
endfunction: sv_csub

function sv_complex sv_cmul(sv_complex c1,
  sv_complex c2);
  return '{c1.re*c2.re - c1.im*c2.im,
    c1.re*c2.im + c1.im*c2.re};
endfunction: sv_cmul

function sv_complex sv_cdiv(sv_complex c1,
  sv_complex c2);
    return '{(c1.re*c2.re + c1.im*c2.im)/
    (c2.re**2.0 + c2.im**2.0),
    (c1.im*c2.re - c1.re*c2.im)/
    (c2.re**2.0 + c2.im**2.0)};
endfunction: sv_cdiv

//other DPI functions
endpackage: sv_complex_pkg
```

Listing 3. Package for defining a complex data type. The package also defines functions for complex calculus, such as complex addition etc. Additionally, our complex calculus package contains some DPI function prototypes. They enable the calculation of the complex elementary functions such as the complex exponential. They are not displayed in the listing.

two values of type `realtime`. They determine the duration of the transaction and the period T_0 of the transformed signal represented by the spectrum.

The content of the spectrum can be obtained in different ways:

- It can be obtained by a transformation of a signal described in the time domain,
- it can be the output of a analog transaction model operating in the frequency domain or
- it can be gained through randomization, which is in general the natural way of gaining stimulus in UVM based testbenches. See an example in listing 4. See also a diagram of the work flow in fig 2.

B. Cubic Spline based stimulus

1) Idea: In the previous subsection, we presented a stimulus generation technique using FOURIER transformation. The

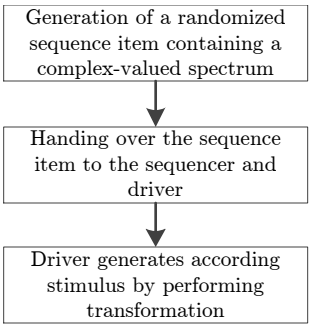

Fig. 2. UVM work flow for generating real-valued stimulus using the FOURIER transformation approach

```
1  `uvm_create(req);
2
3  req.T_realtime          = 2500.0;
4  req.duration_realtime = 10000.0;
5
6  for(int unsigned i=0; i<req.c.size; i++)
7    if(i inside {[5:20]})
8      req.c[i].re =
9        real'($urandom_range(10))/100.0;
10   else
11     req.c[i].re = 0.0;
12
13 for(int unsigned i=0; i<req.c.size; i++)
14   req.c[i].im = 0.0;
15
16 `uvm_send(req);
```

Listing 4. Creating a randomized analog transaction. Note how the the real parts of the frequency portions get randomized in lines 8 and 9.

advantage of this approach is that stimulus can be generated just by randomly selecting some FOURIER coefficients. However, the drawback of this technique is that this kind of stimulus is only useful for designs operating on small signal level. For large signal level designs this kind of stimulus can not be used. Therefore we also developed an approach that allows verification engineers to stimulate such a design using a cubic spline approach supported by UVM.

The key idea of cubic spline interpolation is to interpolate a set of tabulated values in such a way that a smooth signal is resulting. In this context "smooth" means that the second derivative of the interpolation is continuous. See figure 3 for a comparison of linear and cubic spline interpolation.

2) Basic Theory: In this subsection, we briefly explain the basic theory of cubic spline according to [8].

Suppose that some values of a signal $s(t)$ are given in a tabulated way $s_i = s(t_i)$ with $i = 1...N$. To calculate the linear interpolation of the signal $s(t)$ in the interval $[t_j; t_{j+1}]$, one can use the following interpolation formula:

$$s(t) = As_j + Bs_{j+1} \tag{5}$$

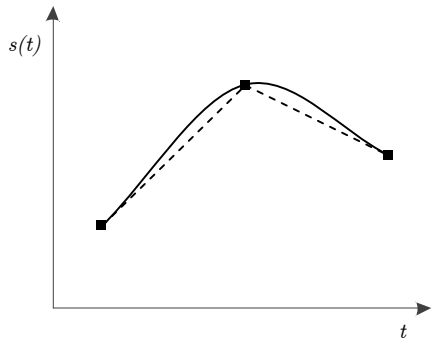

Fig. 3. Comparison of linear and spline interpolation. The points to be interpolated are marked with ■. The linear interpolation is shown as a dashed line.

with

$$A = \frac{t_{j+1} - t}{t_{j+1} - t_j},\ B = 1 - A \tag{6}$$

However, using this formula the result is not smooth. To obtain a smooth result we must extend the formula in such a way that the second derivative is varying linearly through the interval $[t_j; t_{j+1}]$. At the interval's borders the second derivative must reach a value of $\ddot{s}_j$ or respectively $\ddot{s}_{j+1}$ that is supposed to be given.

One can reach that constraints by extending equation (5) in the following way:

$$s(t) = As_j + Bs_{j+1} + C\ddot{s}_j + D\ddot{s}_{j+1} \tag{7}$$

with

$$C = \frac{1}{6}\left(A^3 - A\right)\left(t_{j+1} - t_j\right)^2,\ D = \frac{1}{6}\left(B^3 - B\right)\left(t_{j+1} - t_j\right)^2 \tag{8}$$

The remaining problem is that the second derivative is supposed to be known, but it is not. However, it can be gained by claiming that the first derivative has to be continuous at the interval borders as well. This requirement leads to a linear equation system containing $N - 2$ equations. Its solution are the $\ddot{s}_i$. We will not express the system here. Please refer to [8]. After having the system solved, equation (7) is used to compute a particular value of the signal $s(t)$.

3) Implementation: In order to implement the cubic spline interpolation, we followed the same approach as for the FOURIER transformation: We implemented the algorithm itself in C to ensure fast execution. From the testbench side we are calling a UVM compliant class based interface that hides the implementation using SystemVerilog's direct programming interface.

Also, we implemented a `uvm_sequence_item` and a `uvm_driver` class that enables verification engineers to use the concept of transactions to create randomized stimulus from the test. The parameters in the sequence item are time value pairs that are to be randomized during creation of the item. In SystemVerilog we simple identified these time value pairs using a structure as shown in listing 5.

```
1 package point_pkg;
2   typedef struct {real x; real y;}
3     point;
4 endpackage: point_pkg
```

Listing 5. Package that defines a time value pair type using a `struct`

The creation of a sequence item is shown in listing 6. In line 1 a sequence item is created and its first time value pair is initialized to (0, 0) in line 2. Following that, a random number of values (20 up to 40; line 4) are pushed into the sequence item in a randomized manner (line 8 to 11).

```
1  `uvm_create(req);
2  req.points.push_back(point'('{0.0, 0.0}));
3
4  repeat($random_range(40, 20))
5  begin
6    real old_time = req.points[$].x;
7
8    random_value = (-1.0)**$urandom_range(1)
9      * real'($urandom_range(4000))/1000.0;
10   random_time  = real'($urandom_range(
11     10000000, 500000))/1000.0;
12
13   req.points.push_back(point'('{old_time +
14     random_time, random_value}));
15 end
16
17 `uvm_send(req);
```

Listing 6. Creation of a randomized spline sequence item

The driver is shown in listing 7.

The driver extends `uvm_driver` and therefore integrates into a UVM based testbench environment. The code follows the UVM guide lines. That means that it has a build phase, a connect phase (both not shown in the listing) and a run phase. In the run phase, the driver waits on its according sequencer in order to obtain a sequence item. After having obtained a sequence item, the actual driving takes place in the `drive_transfer` task. There, a wrapper object to the spline interpolation mechanism is created (line 24). This wrapper object has the same purpose as the wrapper objects explained in section IV-A3:

1) It hides the DPI access and
2) allocates the required memory.

After that, the driver calls the spline-method of the wrapper object in order to start the spline interpolation on the C side (line 27). The `while`-loop in line 31 accesses the transformed data using the `get_real` method of the wrapper object and drives the interface variable. How the interface handle is obtained, we will show in the following section.

```
class stimulus_driver extends uvm_driver
  #(stimulus_sequence_item);

//...

// run phase
virtual task run();
  forever begin
    seq_item_port.get_next_item(req);

    $cast(rsp, req.clone());
    rsp.set_id_info(req);

    drive_transfer(rsp);
    seq_item_port.item_done(rsp);
  end
endtask: run

// drive_transfer
virtual task drive_transfer
  (stimulus_sequence_item trans);
  real offset_real = $realtime;

  spline sp = spline::type_id::
    create("sp");

  sp.spline(trans.points);

  #(trans.points[0].x);

  while((offset_real + trans.points[$].x)
    > $realtime)
  begin
    #1;
    vif.signal = sp.get_real($realtime -
      offset_real);
  end
endtask: drive_transfer
endclass: stimulus_driver
```

Listing 7. Spline driver that generates spline-shaped stimulus based upon sequence items containing time-value-pairs

V. Example applications and results

In this section we show some results based upon two examples applications.

First one is small example containing an analog to digital converter (ADC) that we created to test out approach.

The second is a productive motor driver application – e. g. for wipers or electric window lift – from the automotive area.

A. ADC

To have the opportunity to test and refine our approach, we created a model of an ADC. The source code of the ADC is shown in figure 8.

```
module adc
  #(parameter bit_p     = 9,
    parameter ana_max_p = 5.0)

  (input  bit                clk_i,
   input  real               ana_i,
   output bit [bit_p-1 : 0] dig_o);

  real resolution_real =
    2.0 * ana_max_p / real'(2**bit_p - 1);

  always @(posedge clk_i)
  begin
    if(ana_i <= -ana_max_p)
      dig_o = '0;
    else if(ana_i >= ana_max_p)
      dig_o = '1;
    else
    begin
      dig_o = 0;
      for(real i = -ana_max_p; i < ana_i;
        i+=resolution_real)
        dig_o++;
    end
  end
endmodule: adc
```

Listing 8. Real-number-model of an ADC

The ADC samples the real-valued input signal based on the rising edge of the input clock `clk_i`. The parameter `ana_max_p` defines the maximum input swing that the ADC can handle. An input value on `ana_i` that is equal or smaller than -`ana_max_p` will result in an output at `dig_o` of all zero. An input value that is equal or greater than +`ana_max_p` will result in an output value of all one. All values between -`ana_max_p` and +`ana_max_p` will result in a respective digital output whose precision is determined by `bit_p`.

The analog input of the ADC was connected to a verification component (VC) containing our drivers presented in the previous sections. The connection was done via a virtual interface (vi; see listing 9, 10 and 11). In order to push down the handle on the interface to the OVC, the new `uvm_config_db` mechanism is used. Compared to the old OVM configuration mechanism, this new mechanism is much more flexible. The old mechanism was only able to configure variables of type `ovm_bitsteam_t`, `string` or `ovm_object`, whereas the new mechanism can configure variables of any type including virtual interfaces. This is due to the fact that the new mechanism uses a parameterized configuration table that is globally visible.

Another very good feature of the new mechanism is that it explicitly allows to call the `set` method from a non-`uvm_component` context. This is especially useful for push-

ing down virtual interface handles, since interfaces are static and therefore are always instantiated outside the class-based OVM/UVM context.

```
interface real_if;
  real signal;
endinterface: real_if
```

Listing 9. SystemVerilog interface that is used to connect the DUT with the VC. The interface contains only one real-typed signal that will be connected to the ADC's input. The driver will be connected to this signal using a virtual interface handle.

```
initial
uvm_config_db#(virtual real_if)
  ::set(null, "uvm_test_top.env.agent1.*",
  "vif", top.ana_i_if_i);
```

Listing 10. Assignment of the interface to the virtual interface of the VC using the new UVM configuration style `uvm_config_db`. The first argument of the `set` method points to a `uvm_component`. Since the method is called in a `module` context, it can not be set to a meaningful context and is therefore set to `null`. The second argument points to a `uvm_component` relative to the first argument. Since the first argument is `null` in our case, the second argument is the absolute path to the target. The third argument is the name of the field to be set. The fourth argument is the physical interface, whose handle is to be pushed to the VC.

```
function void connect();
  if(!uvm_config_db#(virtual real_if)
    ::get(this, "", "vif", vif))
  begin\
    `uvm_fatal(get_type_name(),
    {"No VIF set for ",
    get_full_name(), "!"});
  end
endfunction: connect
```

Listing 11. The driver and monitor obtain the handle to the interface in the connect phase using the UVM configuration mechanism. In order to do that, the `get` method of the `uvm_config_db` mechanism is used. The syntax is the same as in 10. Additionally, a check is performed, whether the `set` method has been called before. If not, a `uvm_fatal` is issued.

The digital output was connected to a passive VC. Both VCs, we connected to a `uvm_scoreboard` using TLM connections (See figure 4). It receives exactly the same sequence item as the driver inside the active VC. Therefore, it calculates directly with the sequence item's parameters without transforming them to a signal level. We show in another paper, how the scoreboard works.

B. Motor driver

After testing and refining our approach using the ADC as described in the previous section, we applied it in the chip level verification of a product.

This product is a system on chip (SoC) from the automotive area. It is used to drive motors, e. g. in wiper or electrical window lifting applications. Its structure is shown in figure 5.

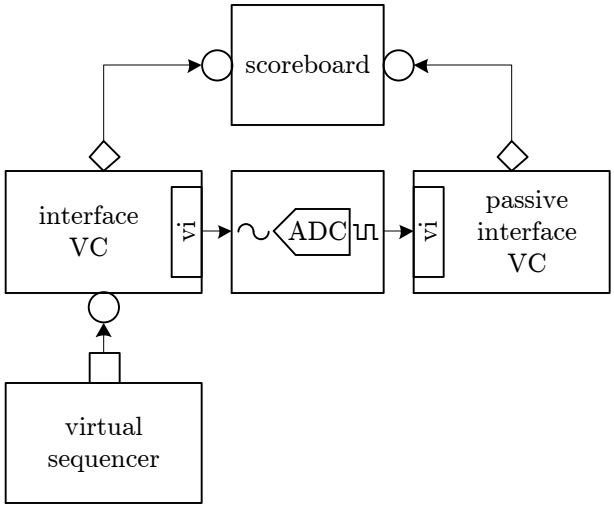

Fig. 4. Testbench structure for an ADC model containing an active and a passive VC as well as a scoreboard

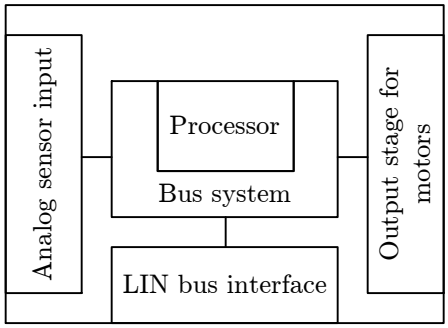

Fig. 5. Structure of the automotive SoC

We connected the analog sensor input shown in fig. 5 with the VC that uses the presented approach, in order to stimulate the interface randomly. The interface has been connected to the VC using the same way as described in the previous subsection.

With the aforementioned approach we were able to find a bug in the RTL code of a state machine in the design. The state machine got stuck after we injected several random spikes into the sensor input using the cubic spline approach. During the verification of that project, we considered the main advantages of our approach that relatively complex stimulus could be generated much faster as with a directed approach.

VI. Conclusion and Outlook

In this paper we gave a brief overview on an abstraction technique and highlighted its key features. The technique tackles the necessity of being able to generate constraint random stimulus also for analog inputs of a DUT. Our future work will focus on the extensions of the presented methodology, regarding usability and flexibility. The goal is to provide a UVM based building box that covers the need of verification engineers to simulate and verify mixed signals designs. This building box shall include methods and techniques for moni-

toring and checking of analog signals, as well as for coverage collection.

REFERENCES

[1] Accellera, *OVM User Guide—Version 2.1.1*, http://www.ovmworld.org, March 2010.

[2] ——, *Universal Verification Methodology (UVM) 1.0 User's Guide*, http://www.uvmworld.org, February 2011.

[3] OSCI, *OSCI SystemC-AMS extensions*, www.systemc-ams.org, March 2010.

[4] N. Khan, Y. Kashai, and H. Fang, "Metric Driven Verification of Mixed-Signal Designs," in *proceedings of DVCon*, March 2011.

[5] T. Butz, *Fourier Transforms for Pedestrians. (Fouriertransformation für Fußgänger) 4th Revised and Expanded ed.* Tuebner, 2005.

[6] M. Frigo and S. G. Johnson, "Fastest fourier transformation in the west," www.fftw.org.

[7] IEEE 1800-2009, *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language*, December 2009.

[8] W. H. Press, T. S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.