2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Verification IP for Complex Analog and Mixed-Signal Behavior

Thilo Vörtler, COSEDA Technologies GmbH, *thilo.voertler@coseda-tech.com*

Karsten Einwich, COSEDA Technologies GmbH, *karsten.einwich@coseda-tech.com*

***Abstract*—This paper presents a framework for the construction of mixed signal verification IP based on SystemC AMS. These verification IP allow the specification of complex analog continuous time behavior, which is often part of standard specifications. In this paper it is demonstrated how signals for analog checks are sampled, arbitrary reference signal values can be described, and custom comparison algorithms can be implemented.**

Keywords—Verification IP;analog mixed-signal; SystemC AMS

I. Introduction and Related Work

The SystemC AMS standard [1][2] allows it to model complex mixed signal systems, including digital and analog behavior, at high abstractions levels. These designs can be used as virtual prototype of a system e.g. for software development, or as golden reference for design implementation in hardware development. For the verification of such systems simulation based verification methodologies like the Universal Verification Methodology [3] can be used. However, these methodologies are tailored towards checking digital behavior, i.e. monitored values are control signals, whose timing can be expressed as multiples of clock edges. Therefore assertion languages like System Verilog Assertions [4] can be used that allow it to specify digital protocol behavior.

In contrast analog signals are characterized by a continuous time scale and violations are characterized in the simplest case by violating a signal threshold that can occur at any time. A threshold is often described as an analog function e.g. a sine wave. Furthermore, checks are often more complicated than detecting a simple threshold crossing. For example analog characteristics like a frequency drift have to be considered within the check algorithm.

Related work in the field of SystemC AMS verification of analog behavior is limited. In [5] an assertion based approach to SystemC AMS verification is given. However, no details about the library implementation are provided. Similarly, in [6] the analog assertion language STL is presented, which however has no implementation available for SystemC AMS. Our work in contrast, is not intended to build assertions as no logical layer is provided to connect checkers. However, the proposed framework allows it to easily describe analog reference functions in the form *f(t)*.

The objective of this paper is to introduce a checking framework that is flexible enough to support the construction of reusable analog verification IP, which can be used within simulation based verification methodologies. The paper is structured as follows. In Section II the general checking framework is introduced. Section III then describes how custom check algorithms can be implemented within the framework to build a library of custom checker verification IP. Section IV will demonstrates how complex analog behavior can be expressed for an operational amplifier circuit in combination with a high pass. In Section V conclusions are given.

II. Checker Framework

A. Motivational Example

As an example for the checker framework the sinusoidal signal show in Figure 1 is used, which can be easily described using SystemC AMS is shown. For the signal (diamond markers) it shall be checked, that it is always below an upper bound reference signal with a certain offset. A check of this kind can be specified using the checker framework using the following syntax:

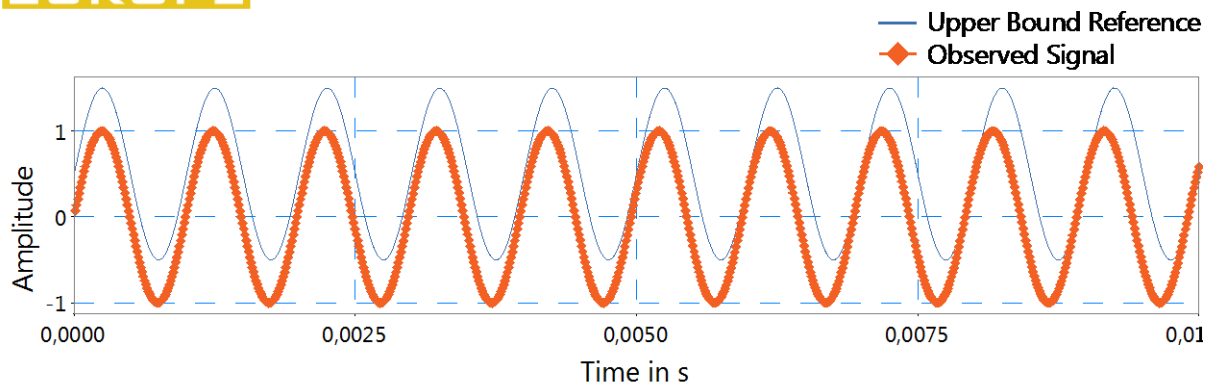

Figure 1: Checking upper bound for a sinusoidal signal compared to a reference signal.

```
CHECK_UPPER_BOUND_DYNAMIC(signal_s, sin_reference(0.5, 1e3), sc_time(0.0, SC_MS),
sc_time(10, SC_MS));
```

Whereby **CHECK_UPPER_BOUND_DYNAMIC** defines that the checker is an upper bounds checker. The observed signal is called **signal_s**, which is passed as a reference. The next argument specifies the reference function that is used as upper bound for the signal. In this case it is sinus function with an offset on the amplitude of 0.5 and a frequency of 1 kHz. Afterwards, the start and end time of the checker is specified in this case checker runs from now up to 10 milliseconds in the future. A checker function can be called within every SystemC method or thread and will be started, when the code location is executed.

In the example it can be seen, that there is a slight frequency mismatch between the observed signal and the reference. The checked signal crosses the upper threshold at about 7 milliseconds simulation time and an error will occur. These errors can be printed on the console, used for coverage collection, or to stop the simulation in case of a regression run.

B. Framework Overview

Every check that a user can specify within the checker framework consists of three basic building blocks. These blocks are *signal preprocessors*, *compare algorithms*, and *references* for expected values. From those a library of building blocks can be created, which can be combined by the user in a kind of "Lego" fashion to form a user specified checker as shown in Figure 2 . A short description of the components follows.

- *Signal preprocessors*: Generates the samples which are used within checkers to compare values with a reference signal. For each time point, which is relative to the start time of the checker, one sample is generated. Samples can be extracted from SystemC AMS signals using *sample generator* classes provided by the framework, or be read from an input files (see Section II.C for details on sampling semantics). It is possible to combine samples from different signals, extracted at different time points, e.g. to build a preprocessor which calculates the difference of two signals.

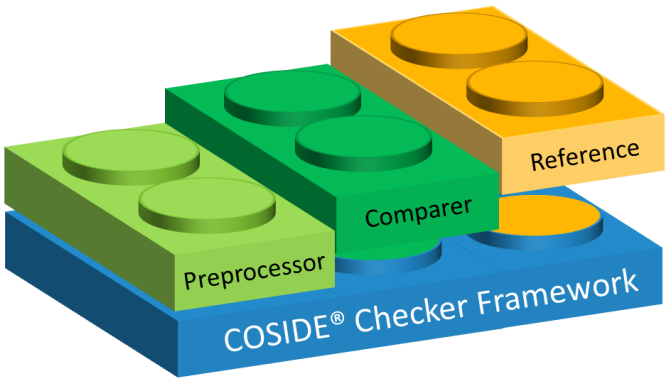

Figure 2: Basic building blocks for creating check functions from a library of building blocks.

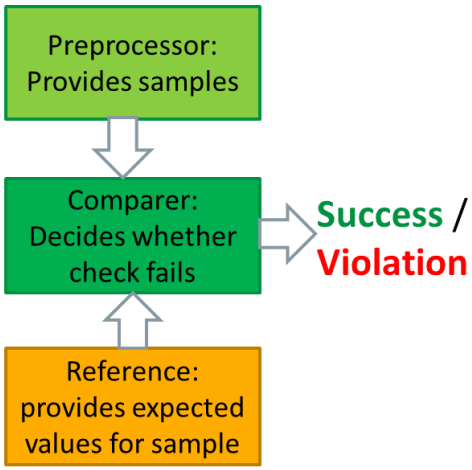

Figure 3: Interaction between the different checker components.

The signal preprocessor is the only part of a checker that is allowed to consume time. It has to provide samples from the specified check function start time till the end time.

- *Compare algorithms*: The basic functionality of the compare algorithm is to decide whether the generated sample value received from the preprocessor is correct compared to reference values. The compare algorithm is called with a sample value and an associated time point to which the sample belongs. As check algorithms are objects they can contain arbitrary complex computations, as they can store values received from previous calls.
- *References*: Provide expected values for a specified time point. The reference values can be either calculated or read from a file source. Depending on the kind of used compare block, also several reference values can be provided for a time point. This is for example useful when the reference specifies an upper and lower bound signal bound. The simplest form of a reference function returns a constant value, independent of the specified time point.

As the comparer is passed the reference calculation object it can request arbitrary many reference values at different time points. Furthermore, a comparer can store sample values for more complex checking algorithms in a local history-

The principle interaction between the building blocks when running a checker is depicted in Figure 3. The preprocessor provides samples, which are passed to the comparer. Based on the sample time the comparer then requests reference values. All computations are triggered by the preprocessor, when a new sample is generated and no delay is allowed in the chain between preprocessor and reference, so that evaluation is immediate. When sample values need to be reused they have to be stored either in the preprocessor (for interpolating preprocessors) or in the comparer for complex checking algorithms.

C. Sampling Semantics for SystemC AMS Signals

As part of the preprocessor the samples used for checking can be extracted from the digital event based SystemC signals, or from calculation points within the SystemC AMS domain, e.g. when an equation is solved or the dataflow cluster calculates new values.

Three kinds of checkers, depending on the time point when they are executed, are differentiated. *Immediate checkers* sample signals within the same SystemC delta cycle of their creation. *Delayed checkers* run at a specified time point in the future. *Dynamic Checkers* are specified with start and end times, relative to their creation time, and can generate multiple samples for checking. An overview of the used sampling semantics to extract values with preprocessors is given in Table 1. The sampling in case of the *immediate* and *delayed* checkers always happens in the last delta cycle of the specified time point.

More complex is the sampling for the dynamic checkers. In case of event driven SystemC signals, a sample is generated when an event on the signal occurs during the runtime of the checker. As in this case for constant SystemC signals no sample would be generated, the signal is always sampled at the beginning of the interval.

Table 1: Sampling semantics for SystemC AMS signals.

Checker type	SystemC event driven signals	SystemC AMS signals (TDF, LSF, ELN)
Immediate checker	Signal is sampled in the same delta cycle.	Signal is sampled in the same delta cycle and no interpolation is performed.
Delayed checker	Signal is sampled at the specified time point using the last delta cycle value of the checked signal (last visible signal value).	Signal is sampled at the specified time point and no interpolation is performed.
Dynamic checker	Signal is sampled at each SystemC time point , for which an event occurs within the specified time range. The sampled value is the value of the last delta cycle (last visible signal value). A sample is always generated at the start point of the interval.	Signal is sampled at each SystemC AMS calculation time point in the specified time range.

In case of SystemC AMS signals values are updated at the calculation time points of the AMS cluster. Therefore SystemC AMS has been extended with additional callbacks that allow it to create events when new values in a cluster are generated. Thereby it is also possible to react on dynamic TDF time steps as sampling is based on the cluster calculation time points. Whenever a new calculation is performed a new sample is generated.

From the user point of view all calls to the user checker functions are non-blocking i.e. they return in the same delta cycle, without any time passing. Therefore, the framework automatically spawns processes in case of the *delayed* and *dynamic checker* functions.

D. Additional Framework Features

In addition to providing base classes for implementing the checker building blocks the framework provides features for regression runs and debugging of failed checks:

- All checkers are automatically registered within a central registry for creation of post simulation summary.
- Logging of checker statistics including number of failed checks, checked samples, which can be used for coverage collection.
- User definable, SystemC based severity mechanism for checkers, allowing it to stop simulation when a check fails. Log messages can be filtered using standard SystemC mechanism.
- Debugging of failed checks using automatic tracing of samples and reference values in a file as shown in Figure 1.
- A library of pre-defined check components like differential signal preprocessors, bound based compare algorithms (upper, lower, range) and reference functions.

III. BUILDING CUSTOM VERIFICATION IP

The proposed framework provides base classes and utility functions for implementing custom building blocks. This section demonstrates how these can be used to create a library of checker verification IP. A user of the verification IP can call a dynamic checker (running in a specified time range) with arbitrary library blocks, using the following syntax:

```
CHECK_DYNAMIC(preprocessor(arguments), comparer(arguments), reference(arguments),
start_time, end_time);
```

Typical arguments for the preprocessor are the SystemC signals used for sampling. For the comparer and reference optional arguments for configuration of the blocks can be passed.

One challenge for implementing the blocks, is to make them generic by using C++ templates, so that different SystemC datatypes can be used. In the following section, based on code examples, the implementation of the building blocks using the APIs provided by the checker framework is shown.

A. Preprocessor Multiplying Signal With Constant Factor

```
1  template<class Tsignal, class Tsample>
2  class factor_preprocessor: public check_pre_processor_base<Tsample> {
3  public:
4    factor_preprocessor(Tsignal& signal, Tsample factor) : factor(factor) {
5                  this->register_sample_gen(new sc_gen<Tsignal>(signal));
6    }
7    void operator() (const sc_core::sc_time& sample_t, const Tsample& sample) override {
8      this->sample_valid(sample_t, sample * factor);
9    }
10 private:
11   Tsample factor;
12 };
```

This example demonstrates how a preprocessor that multiplies samples received from a SystemC Signal with a constant factor is implemented. A custom preprocessor is derived from the template base class **`check_pre_processor_base<Tsample>`**, whereby the template parameter **`Tsample`** defines the type of samples the preprocessor produces. The constructor (line 4) takes as argument the SystemC AMS signal, which shall be observed and a constant factor for multiplication of the sample values. In line 5 the signal is registered within the checker framework using the **`register_sample_gen`** method using a predefined sample generator. A sample generator extracts samples from SystemC AMS signals as described in Table 1. Thereby all SystemC and AMS signal and port types are supported. It is also possible to sample values from multiple signals, by registering several sample generators with corresponding processing callbacks.

The actual function the user has to implement is the **`operator()`**, which is called by the checking framework whenever a new sample is available on the registered signals. The extracted sample can then be modified with an arbitrary algorithm creating one final sample, e.g. samples from different sources can be added. The resulting sample is passed to the comparer using the function **`sample_valid`**, which takes as argument a sample of type **`Tsample`** and the corresponding sample time. It is also possible to register multiple sample generators as source for a sample

B. Upper Bound Compare Algorithm

```
1  template<class Tsample, class Tref>
2  class check_upper_bound_comp: public check_comp_base<Tsample, Tref> {
3
4    bool operator()(const sc_core::sc_time& checker_time,
5                    const Tsample& value, check_ref_base<Tref>& ref) override {
6      last_sample = value;
7      last_ref = ref(checker_time);
8      return (last_sample <= last_ref);
9    }
10 private:
11   Tsample last_sample;
12   Tref last_ref;
13 };
```

A custom compare algorithm is implemented by deriving from the class **`check_comp_base<Tsample, Tref>`**, whereby the template parameters define the type of the samples and reference values the comparer accepts. The user has to implement the **`operator()`**, which is passed the time point of the sample, the sample value and a reference function object. The function returns **`bool`** signaling whether the check was successful or failed. In line 7 the reference function is called with the current checker time, to access a value from the reference. A comparer can request can values from the reference at different time points for more complex calculations. The actual upper bound comparison is performed in line 8 and the return value is determined.

C. Sinus reference function

```
1  class sin_reference: public check_ref_base<double> {
2  public:
3
4    sin_reference(double offset, double freq): offset(offset), freq(freq){}
5
6    const double& operator ()(const sc_core::sc_time& tm) override {
```

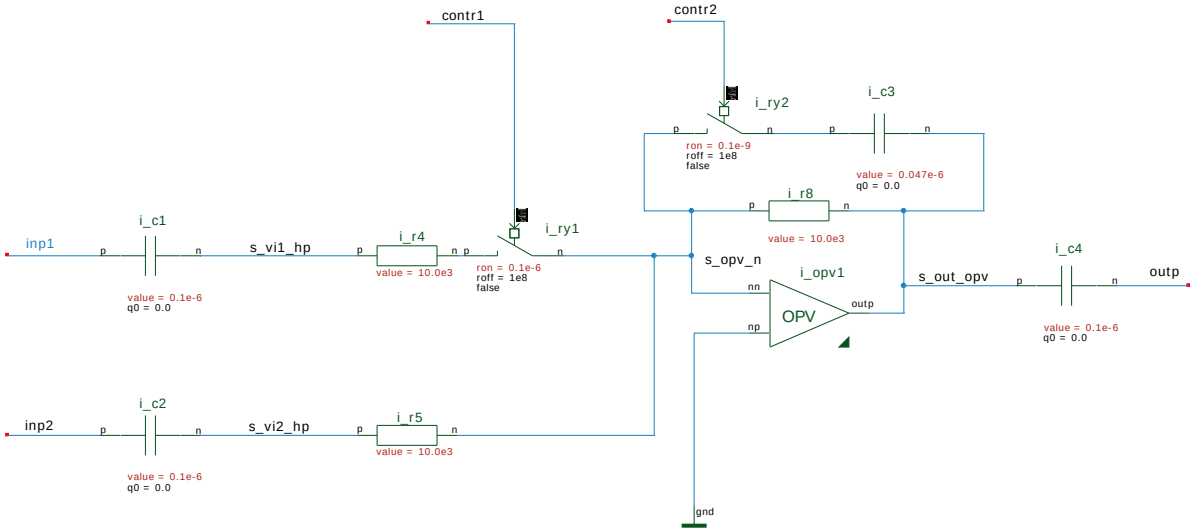

Figure 4: op-amp circuit for adding analog input voltage signals

```
 7      // Calculation of the new sinus value
 8      current_value = sin(2.0*M_PI*freq*tm.to_seconds()) + offset;
 9      return current_value;
10    }
11
12  private:
13    double current_value;
14    double offset,freq;
15  };
```

Similar to the preprocessor and comparer, a custom reference function is derived from the base class **`check_ref_base<Tref>`**. In this case the reference function provides samples of type **`double`**. Again the operator() has to be implemented by the creator of the reference block, which takes as argument the time of the expected reference value. For this reference an offset and frequency are specified within the constructor (line 4). The actual calculation of the sinus value, depending on the arguments is performed in line 8.

IV. Application Example – OP-AMP with Fault Injection

To demonstrate the flexibility of the framework for the verification of an analog applications consider the op-amp circuit shown in Figure 4. This is a typical circuit, as it can be simulated using the electric linear network (ELN) computation model of SystemC AMS. The basic operation of this circuit is adding two analog input signals **`inp1`** and **`inp2`** with the result being signal **`outp.`** Each of the input signals is dc decoupled using a high pass filter. Using the digital control signal **`contr1`** it can be chosen, whether **`inp1`** shall be added to the signal. In addition the circuit supports a mode, where an additional feedback capacitor can be activated (signal **`contr2`**, low pass behavior).

The expected behavior of the circuit can be calculated from the equations of a first order high pass and the op-amp for a sinusoidal signal, when disabling the feedback capacitor (**`contr2 = false`**). The output amplitude of a first order high pass is:

$$U_{out} = \frac{\omega CR}{\sqrt{1+(\omega CR)^2}} * U_{in} \tag{1}$$

Similarly the phase response can be calculated as:

$$\varphi(\omega) = \arctan\left(\frac{1}{\omega CR}\right) \tag{2}$$

To easily implement this specification a generic reference function was implemented (compare Section III.C) that allows the expression of an arbitrary function *f(t)*. This reference function takes as constructor argument a C++11 lambda function, which describes the reference behavior. A checker with the reference behavior can thus be implemented in SystemC AMS as shown below:

```
1  // Define sinus input lambda function
2  auto f_inp1=[=](double t){ return sin(2.0*M_PI*f1*t); };
3  auto f_inp2=[=](double t){ return sin(2.0*M_PI*f2*t); };
4
5  // Amplitude of high pass (wCR)/sqrt(1.0 + (wCR)^2 )
6  double a_vinp1 = 2.0*M_PI*f1*c1*r4/sqrt((1.0+pow((2.0*M_PI*f1*c1*r4),(2.0))));
7  double a_vinp2 = 2.0*M_PI*f2*c2*r5/sqrt((1.0+pow((2.0*M_PI*f2*c2*r5),(2.0))));
8
9  // Phase of high pass arctan(1.0/(wCR))
10 double ph_hp_inp1 = atan(1/(2.0*M_PI*f1*c1*r4));
11 double ph_hp_inp2 = atan(1/(2.0*M_PI*f2*c2*r5));
12
13 // Check result
14 CHECK_RANGE_EPS_DYNAMIC(outp,
15                 f_t([=](double t)   //reference (expected) function
16                                 { return -(  a_vinp1*f_inp1(t+ph_hp_inp1/(2.0*M_PI*f1)) +
17                                              a_vinp2*f_inp2(t+ph_hp_inp2/(2.0*M_PI*f2)));
18                                 }) ,
19                                 0.1, sc_time(1.0, SC_MS), sc_time(4.0, SC_MS));
```

In lines 1-3 the input signal behavior is defined for the two inputs **`inp1`** and **`inp2`**, whereby **`f1`** and **`f2`** describe the chosen frequencies of the signals. Similarly in lines 5-7 and lines 9-11 the output amplitude and the phase response of the high pass are calculated according to equations (1) and (2).

The actual definition of the checker for the signal **`outp`** is shown line 14-19. The checker is a range checker, which uses for comparison an absolute epsilon range of 0.1 (line 19) around the defined signal reference. The reference function is passed a lambda function (lines 15-18), which calculates the (negated) sum of the two input signals. In Figure 5 simulation results for an input frequency of f1 = 2.73 kHz and f2 of 1 kHz are shown. At the top output signal **`outp`** is shown, with the allowed boundaries calculated from the reference function. Below the corresponding input values are shown. It can be seen that the output signal stays within the specified range.

For demonstrating the error detection capabilities a fault was injected in the op-amp circuit. Therefore, an additional resistor was added at **`inp1`**. This resistor leads to a different behavior of the input high pass, corresponding to a deviation from the system specification. As the reference is unchanged the checker framework is able to detect

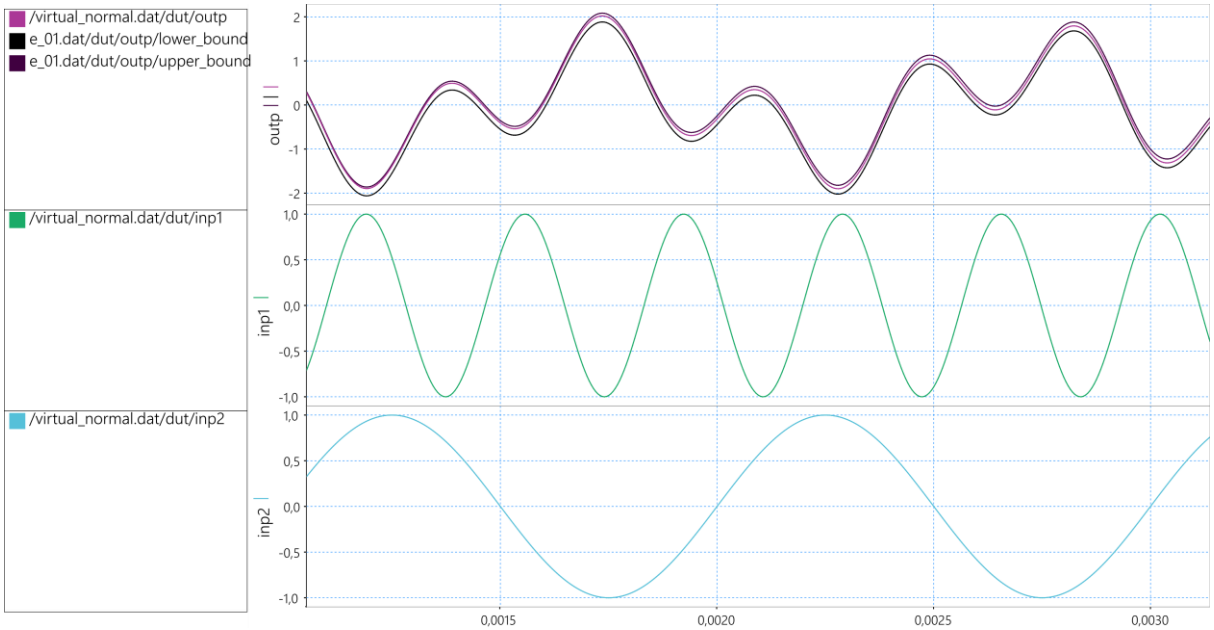

Figure 5: Simulation results of the op-amp with reference values calculated using the checker framework.

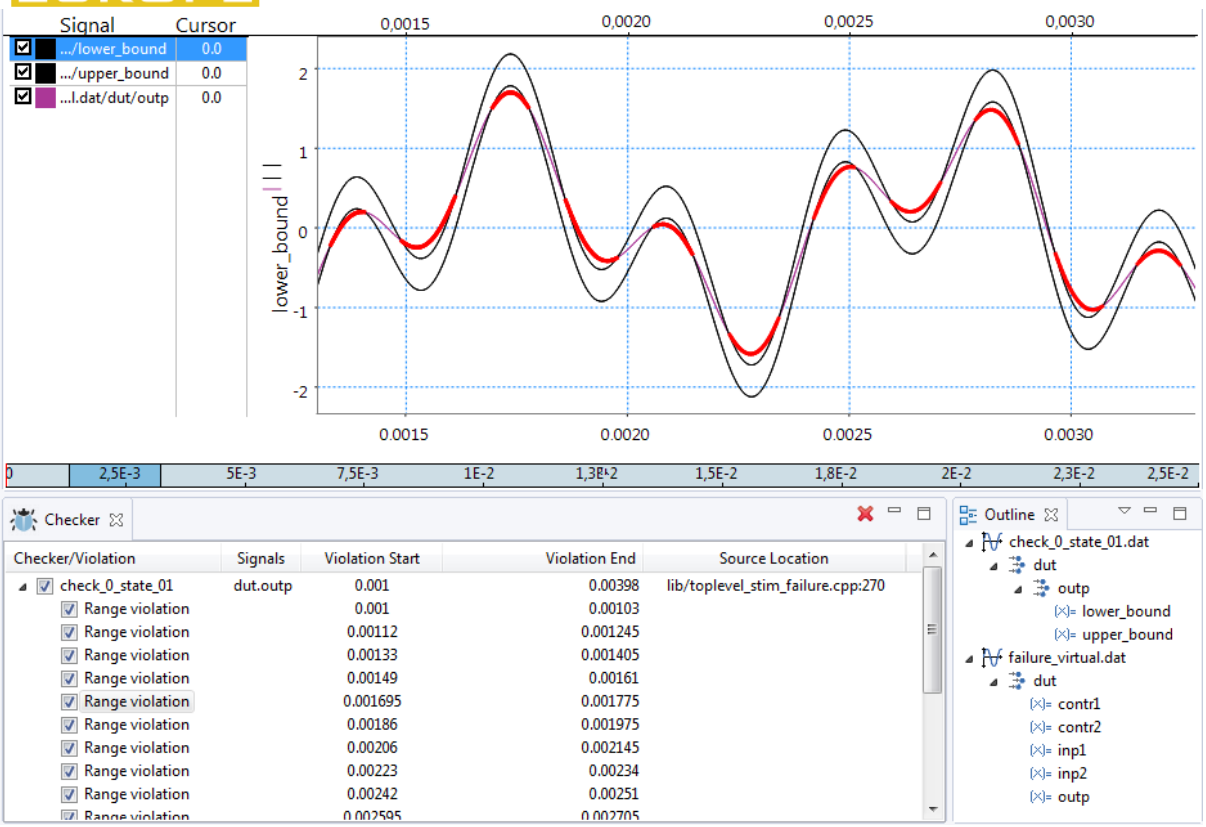

Figure 6: Simulation result for failing checker within the COSIDE® tool environment [7].

theses errors, without any change on the verification part. The simulation results with the added resistor are shown n Figure 6. The allowed boundaries are shown in black as in Figure 5. When the signal violates the defined boundaries the output signal is marked red. The current tool integration within the COSIDE® tool environment from COSEDA technologies allows it to graphically highlight the time points when checker violations occurred, select violation points and to collect statistics.

V. CONCLUSIONS

This work presents a verification framework for the construction of custom analog verification IP for SystemC AMS. As shown in the example it is possible to easily describe expected complex analog behavior as in a specification document and independently from a SystemC AMS model of computation. With these checks it is for example possible to automatically detect errors during fault simulation or regression runs. Furthermore, checks can be created dynamically and can also be easily integrated in complex verification environments like UVM SystemC.

REFERENCES

[1] "IEEE Standard for Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual," in IEEE Std 1666.1-2016, vol.,no.,pp.1-236, April 6 2016 doi: 10.1109/IEEESTD.2016.7448795

[2] "IEEE Standard for Standard SystemC Language Reference Manual," in IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) , vol., no., pp.1-638, Jan. 9 2012 doi: 10.1109/IEEESTD.2012.6134619

[3] "Universal Verification Methodology 1.2 Class Reference," *Accellera System Initiative* (2014).

[4] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009) , vol., no., pp.1-1315, Feb. 21 2013 doi: 10.1109/IEEESTD.2013.6469140

[5] Lämmermann, Stefan, et al. "Checking heterogeneous signal characteristics applying assertion-based verification." *Frontiers in Analog Circuit Verification-FAC* (2009).

[6] Nickovic, Dejan, and Oded Maler. "AMT: A property-based monitoring tool for analog systems." *Formal Modeling and Analysis of Timed Systems* (2007): 304-319.

[7] „COSIDE® 2.4", Coseda Technologies GmbH, http://www.coseda-tech.com/ .