Verification IP for Complex Analog and Mixed-Signal Behavior

Thilo Vörtler, *thilo.voertler@coseda-tech.com*
Karsten Einwich, *karsten.einwich@coseda-tech.com*
COSEDA Technologies GmbH

Agenda

- Introduction and Motivation
- Checker Framework for SystemC AMS
- Building Custom Verification IP
- Application Example: OP-AMP with Fault Injection

Introduction and Motivation

- SystemC AMS enables the creation of high level virtual prototypes with analog behavior
- Verification of Mixed-Signal behavior at the system level is challenging
  → Combination of analog (continuous time) and digital (discrete event) domain
- Different cultures of verification:

Digital	Analog
• discrete change of value	• Continuous change of value
• Verification methodologies like UVM	• Custom block level testbenches
• Assertion languages like SVA, PSL	• No standardized verification language
• Highly automatized regression runs	• Manual Waveform debugging

- Goal provide Framework for SystemC AMS that enables automatic checks for analog behavior

Introduction and Motivation

- SystemC AMS uses two timing domains:
  - digital/discrete event: calculated with every signal change
  - analog/cluster period based: calculation points at certain timesteps (constant/dynamic)
- Digital assertions sample at pre-defined sampling points (clock edge)

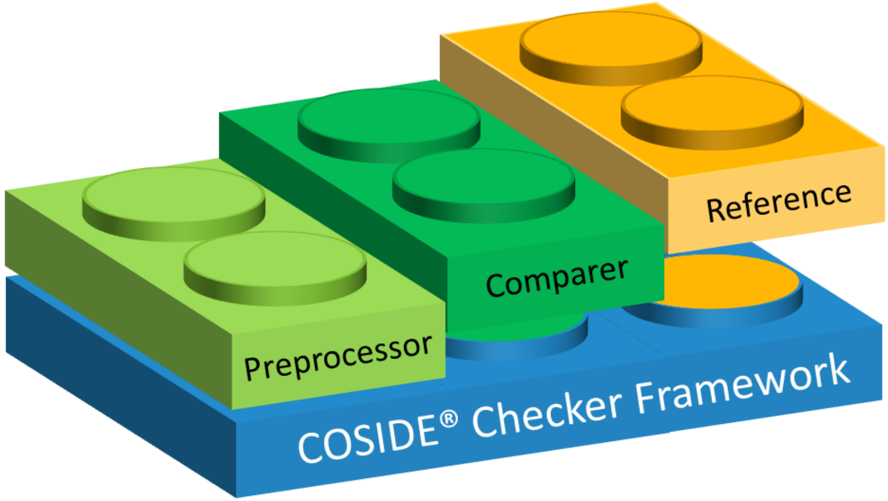
```
assert property (@(posedge clk) sig < 5);
```

- What happens when signal frequency changes? What is the correct sampling frequency?

→Use simulator calculation points for checking and calculation of reference values → continuous check of signal

Introduction and Motivation

- Provides functions for checking of static and dynamic values
- Arbitrary complex analog and digital behavior can be checked
- Checkers build on a “Lego” principle
- Debug and logging features available
- Enables regression testing and test re-use
- Automatic memory management for components

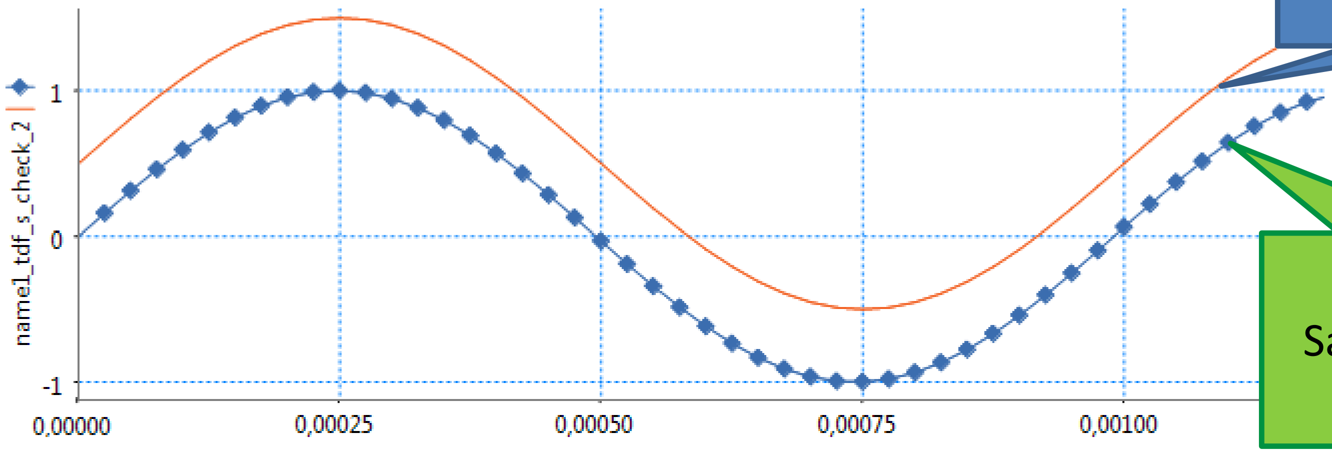


Checker Framework for SystemC AMS

- User checker example

```
// Upper bound check with sinus reference fequency 1khZ
CHECK_UPPER_BOUND_DYNAMIC(name1_tdf_s, sin_reference(1e3), sc_time(0.0, SC_SEC), sc_time(10, SC_MS));
```

Comparer: Upper bound check, valid if signal samples below reference



Reference: Custom defined functor

Preprocessor: Samples of checked signal

Checker Framework for SystemC AMS

- Pre defined checker types:
  - **immediate** checkers, check at the current time point
  - **delayed** checkers, check at a time point after a specified time
  - **dynamic** checkers, check over a time interval specified by start and end time

- Delayed and dynamic checkers spawn own process, all checkers are non-blocking.

- Can be used in testbenches and SystemC modules modules.
- Support all SystemC AMS signal and port types.

Checker Framework for SystemC AMS

Name	Short description
CHECK(target, value, *SC_SEVERITY, name*);	Checks that *target* equals *value*
CHECK_RANGE(target, lower, upper, *SC_SEVERITY, name*);	Checks that *lower* **<=** *target* <= *upper*
CHECK_RANGE_EPS(target, value, eps, *SC_SEVERITY, name*);	Checks that (*value* - *eps*) <= *target* <= (*value* + *eps*)
CHECK_UPPER_BOUND(target, upper_bound, *SC_SEVERITY, name*);	Checks that *target* <= *upper_bound*
CHECK_LOWER_BOUND(target, lower_bound, *SC_SEVERITY, name*);	Checks that *target* >= *lower_bound*

- Perform check in the same delta cycle and return immediately
- An optional **SC_SEVERITY** can be passed → A report with default SystemC severity SC_WARNING is created, when the check fails.
- An optional checker ***name*** can be passed as last argument

Checker Framework for SystemC AMS

Name	Short description
CHECK_AFTER(target,value, time, sc_time_unit, *SC_SEVERITY, name*);	Checks that *target* equals *value* after *time*
CHECK_RANGE_AFTER(target, lower, upper, time, sc_time_unit, *SC_SEVERITY, name*);	Checks that *lower* **<=** *target* <= *upper* after *time*
CHECK_RANGE_EPS_AFTER(target, value, eps, time, sc_time_unit, *SC_SEVERITY, name*);	Checks that (*value* - *eps*) <= *target* <= (*value* + *eps*) after *time*
CHECK_UPPER_BOUND_AFTER(target, upper_bound, time, sc_time_unit, *SC_SEVERITY, name*);	Checks that *target* <= *upper_bound* after time**.**
CHECK_LOWER_BOUND_AFTER(target, lower_bound, time, sc_time_unit, *SC_SEVERITY, name*);	Checks that *target* >= *lower_bound* after time**.**

- Perform check at the specified timepoint and return immediately (without delay, checking is performed in a spawned process)

Checker Framework for SystemC AMS

Name	Short description
CHECK_DYNAMIC (target,value, start_t, end_t, *SC_SEVERITY, name*);	Checks that *target* equals *value* between *start_t* and *end_t*.
CHECK_RANGE_DYNAMIC(target, lower, upper, start_t, end_t,*SC_SEVERITY, name*);	Checks that *lower* **<=** *target* <= *upper* between *start_t* and *end_t*.
CHECK_RANGE_EPS_DYNAMIC(target, value, eps, start_t, end_t, *SC_SEVERITY, name*);	Checks that (*value* - *eps*) <= *target* <= (*value* + *eps*) between *start_t* and *end_t*.
CHECK_UPPER_BOUND_DYNAMIC(target, upper_bound, start_t, end_t, *SC_SEVERITY, name*);	Checks that *target* <= *upper_bound* between *start_t* and *end_t*.
CHECK_LOWER_BOUND_DYNAMIC(target, lower_bound, start_t, end_t, *SC_SEVERITY, name*);	Checks that *target* >= *lower_bound* between *start_t* and *end_t*.

- Perform check in the specified time range and return immediately (without delay, checking is performed in a spawned process)

Checker Framework for SystemC AMS

Checker Examples

- Check that value of signal equals to 1:

```
CHECK(dut->add_value, 1);
```

- Check that value of signal to be after 1us to be between 1 and 2 from current simulation time:

```
CHECK_RANGE_AFTER(dut->add_value, 1, 2, sc_time(1, SC_US));
```

- Check SC_SIGNAL to be between 1 and 2 with severity SC_ERROR from current simulation time up to 0.25 seconds in the future:

```
CHECK_RANGE_DYNAMIC(dut->add_value, 1 , 2,  sc_time(0.0 ,
SC_MS), sc_time(0.25, SC_MS))
```

Checker Framework for SystemC AMS

- The checker library provides utility functions to control the checker status and enable logging and coverage
- A summary can be printed on all the checkers in the system for regression tests

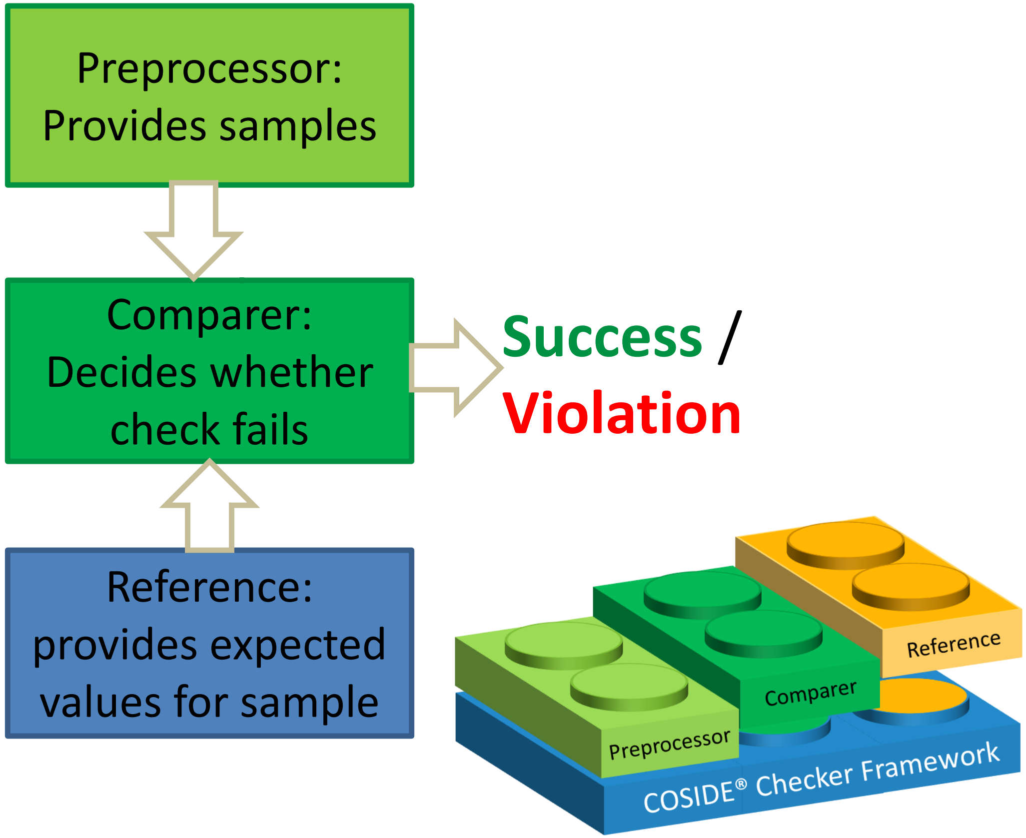
```
-------------------------------------------------------------------------------------------------------
|         Check name|Violation count|   Status|Checked signal|                          Check location|
-------------------------------------------------------------------------------------------------------
|check_0_name1_tdf_s|              0|scheduled|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:74|
|check_1_name1_tdf_s|              0|scheduled|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:78|
|check_2_name1_tdf_s|              0|scheduled|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:82|
|check_3_name1_tdf_s|              0|scheduled|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:86|
-------------------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------------------------
|         Check name|     Violation count|   Status|Checked signal|                          Check location|
------------------------------------------------------------------------------------------------------------
|check_0_name1_tdf_s|                   0| finished|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:74|
|check_1_name1_tdf_s|                   0| finished|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:78|
|check_2_name1_tdf_s|                   2| finished|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:82|
|check_3_name1_tdf_s|                   2| finished|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:86|
|check_4_point_check|                   1| finished|   name1_tdf_s|sinus_check/sinus_check_simple_tb.cpp:95|
------------------------------------------------------------------------------------------------------------
```

Building Custom Verification IP

- Preprocessor
  - Determines the simulation samples used for checking (SystemC, SystemC AMS, File source)
- Comparer
  - Compares expected values with reference values based on a comparison algorithm (e.g. check for upper bound)
- Reference
  - Defines the expected values (Calculated by a function, file source)
- COSIDE® Checker Framework
  - Provides general infrastructure for reporting, tracing, debugging

- User definable building blocks



Preprocessor:
Provides samples

Comparer:
Decides whether check fails

Reference:
provides expected values for sample

Success / **Violation**

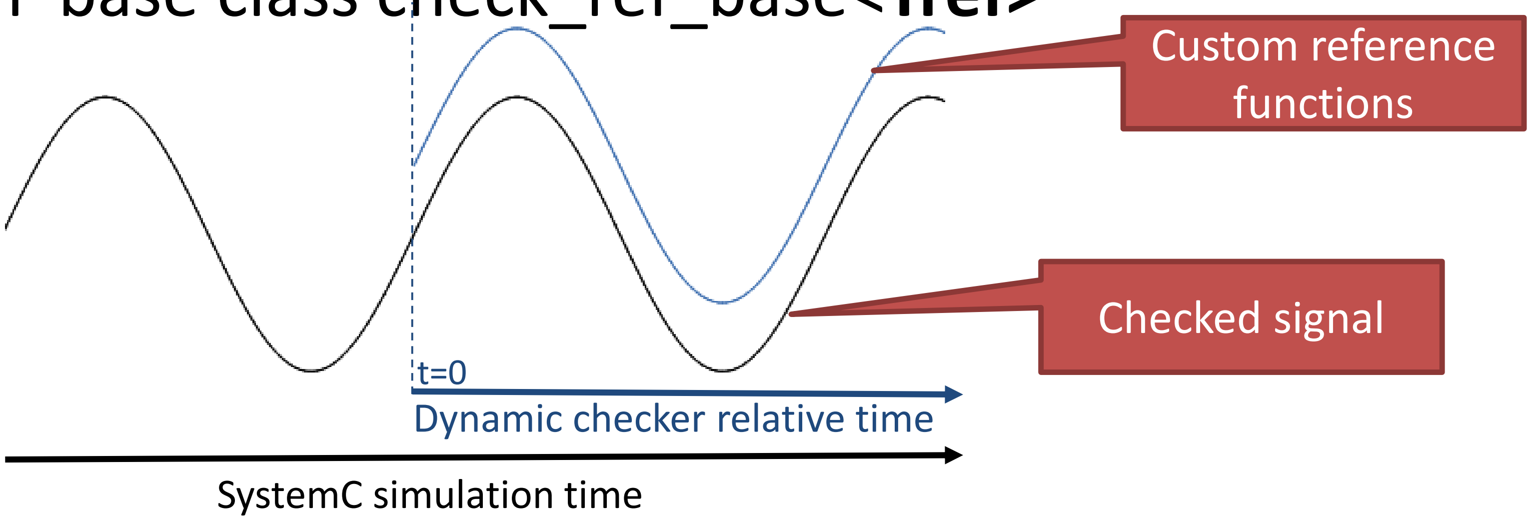
Building Custom Verification IP

Checker Semantics for preprocessors

Checker type	SystemC - Signals	SystemC-AMS
Immediate	Check is executed in the same delta cycle and returns immediately	Check is executed in the same delta cycle and returns immediately → The last calculated (currently valid) AMS value is used
Delayed	Check is executed at the specified time point using the last delta cycle signal value of the checked signal (visible signal value)	Check is executed at the specified time point → The last calculated (currently valid) AMS value at this time point is used
Dynamic	Check is executed at each SystemC calculation time point - using the last delta signal cycle value (visible signal value) – in the specified time range →The signal is always checked at the specified start time point. Further checks depend on the get_sample_event() function of the signal.	Check is executed at each SystemC AMS cluster calculation time point in the specified time range → When no cluster calculation is performed in the time range NO check is performed and a warning with the severity of the checker is printed.

Building Custom Verification IP

- Reference functors allow it to specify arbitrary reference functions and use them together with a checker
- Functor has to implement operator (), which has to return the reference value at the given time **relative** to the the **checker start time**
- Derive from base class check_ref_base<**Tref>**


Custom reference functions

Checked signal

Building Custom Verification IP, Custom Reference

```
// Custom functor passed to the checker, derives from check_ref_base
class sin_reference: public coside_checker_namespace::check_ref_base<double> {
public:
  // Constructor allows to add an offset to the sinus
  sin_reference(double offset): current_value(0), offset(offset) {}
  // Operator which has to be implemented(). Returns the reference value at the given time.
  const double& sin_reference::operator ()(const sc_core::sc_time& checker_time) {
    // Calculation of the new sinus value
    current_value = (sin_function(checker_time, 0, 1e3, 1)) + offset;
    return current_value;
  }
private:
  double current_value;
  double offset;
};
// Sinus function to be used as reference
double sin_function(sc_core::sc_time t, double phase, double freq, double ampl)
{
  double time = t.to_seconds();
  double phase_rad = phase * (M_PI / 180);
  double omega = 2.0 * M_PI * freq;
  double val = std::sin(omega * time + phase_rad);
  return (val * ampl);
}
```

…

Building Custom Verification IP, Custom Reference

```
…
CHECK_UPPER_BOUND_DYNAMIC(name1_tdf_s, sin_reference(0.5), sc_time(1.0, SC_MS), sc_time(2.5, SC_MS), sc_core::SC_WARNING);
```

- Defining custom checkers – Creating Custom Reference Functors
- The checker macros can either use reference functors or static bounds (see previous examples)

Building Custom Verification IP, Custom Comparer

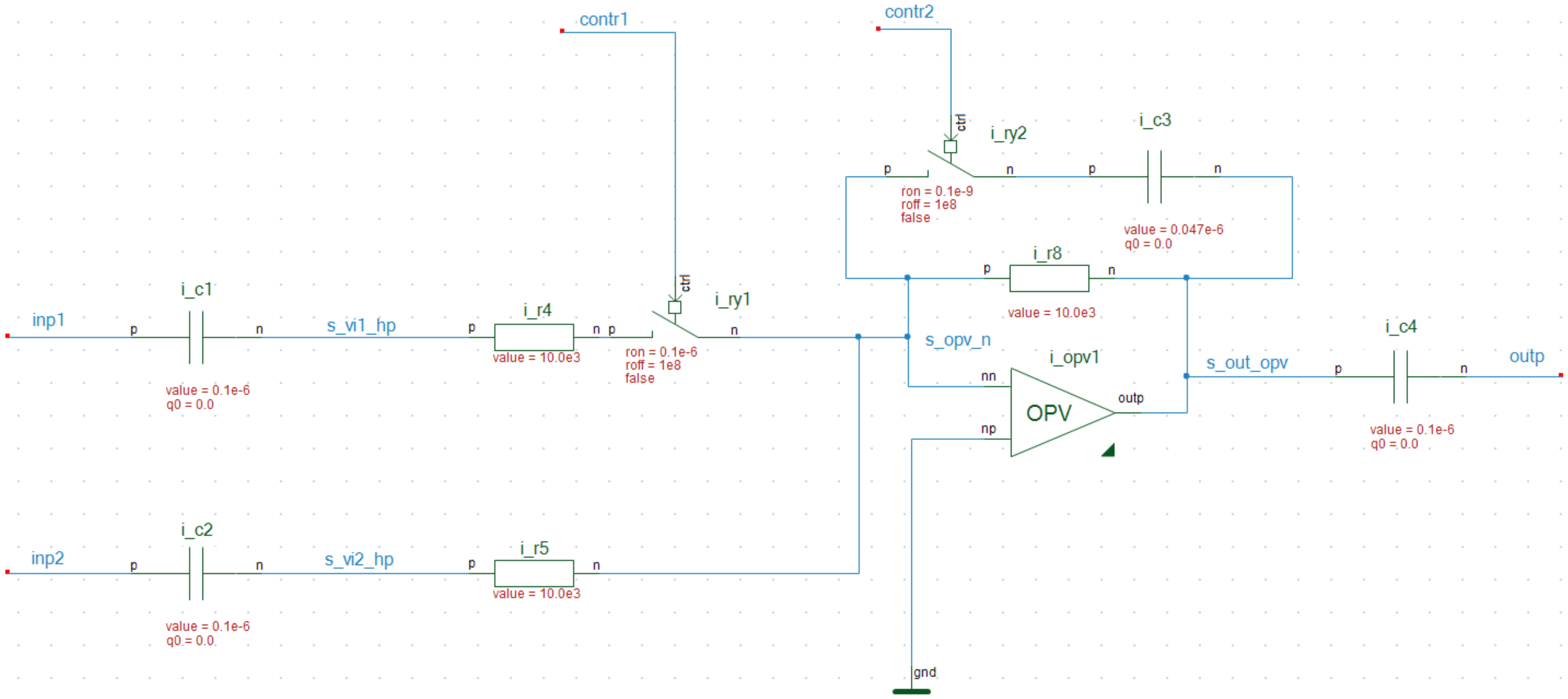
- Derive from base class check_comp_base<Tsample,Tref>
- Has to implement operator **()**, which does the actual checking
- Parameter **`ref`** passes the reference function to be called in the checker
- Optional trace interface for logging values

```
template<class Tsample, class Tref> class check_upper_bound_comp: public check_comp_base<
Tsample, Tref> {

bool inline operator()(const sc_core::sc_time& checker_time,
const Tsample& value, check_ref_base<Tref>& ref) override {
  last_sample = value;
  last_ref = ref(checker_time);
  return (last_sample <= last_ref);
}

private:
Tsample last_sample;
Tref last_ref;
};
```

Application Example: OP-AMP with Fault Injection

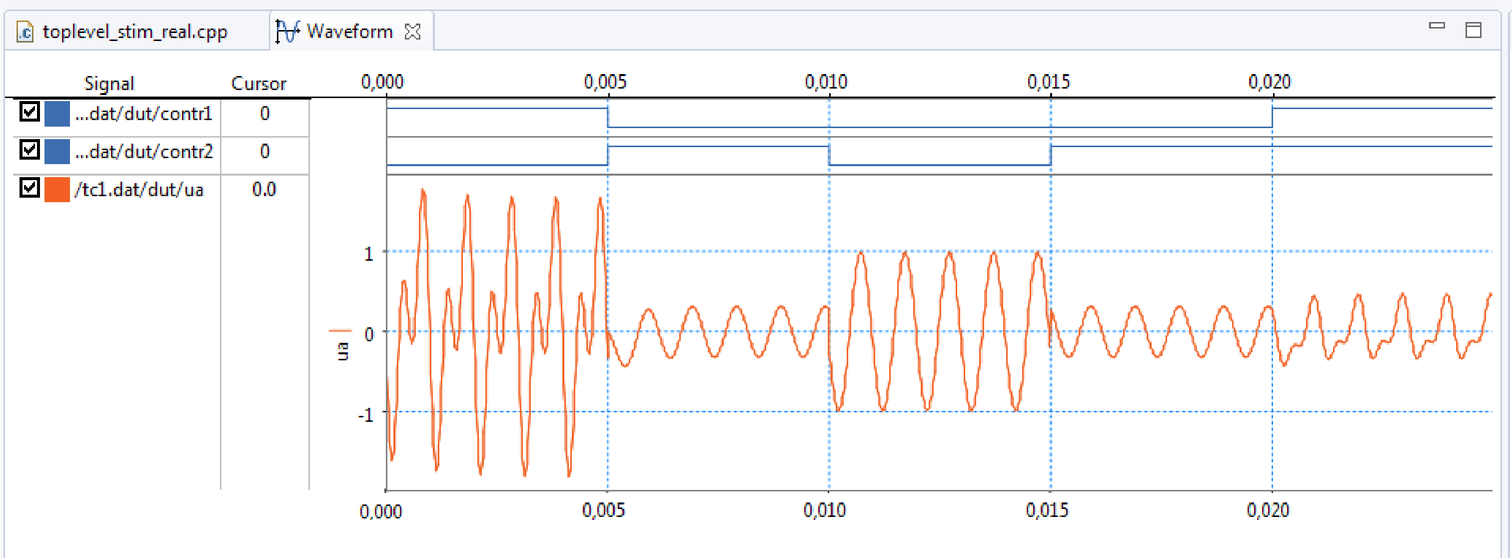


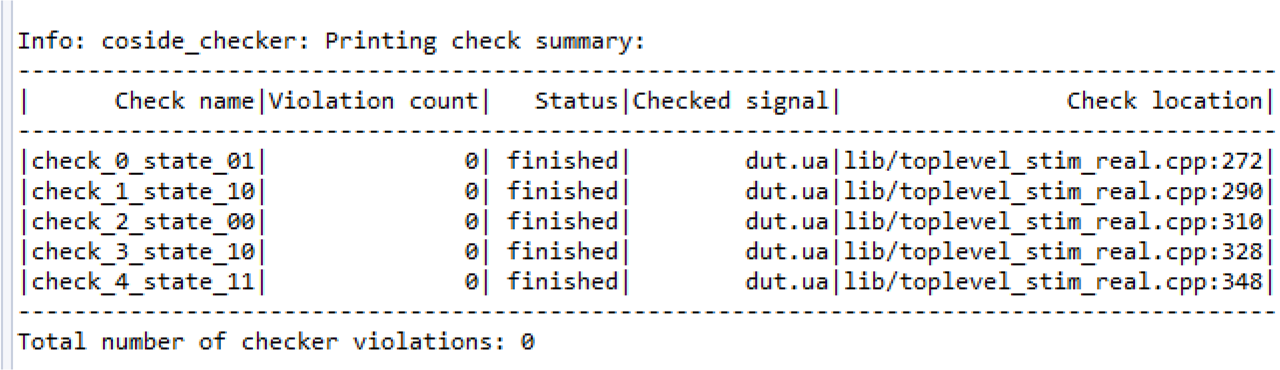
Application Example: OP-AMP with Fault Injection

```
double f1 = 2.73e3;  double f2 = 1e3;
//define function for input signal generation
auto f_ue1=[=](double t){ return sin(2.0*M_PI*f1*t); };
auto f_ue2=[=](double t){ return sin(2.0*M_PI*f2*t); };
//program first configuration
s.ctrl1.set_value(true);
s.ctrl2.set_value(false);
//set analog input signals
s.ue1.set_dynamic(f_ue1);
s.ue2.set_dynamic(f_ue2);
//calculate expected values
//amplitude of high pass (wCR)/sqrt(1.0 + (wCR)^2 )
a_ue1 = 2*M_PI*f1*c_ue1*r_ue1/sqrt((1+pow(2*M_PI*f1*c_ue1*r_ue1,2)));
…
//amplitude of high pass arctan(1.0/(wCR))
phi_ue1 = atan(1/(2.0*M_PI*f1*c_ue1*r_ue1));
…
// check result
CHECK_RANGE_EPS_DYNAMIC(ua,
    f_t([=](double t)   //reference (expected) function
     { return     -(a_ue1*f_ue1(t+phi_ue1/(2*M_PI*f1)) +
                    a_ue2*f_ue2(t+phi_ue2/(2*M_PI*f2)));}) ,
      0.2, sc_time(1.0, SC_MS), sc_time(4.0, SC_MS));
wait(5.0,SC_MS);
```

- Programs digital inports
- Sets analog stimuli
- Calculates expected values
- Checks results with dynamic check that uses a constant epsilon range
- Uses a custom reference function that takes a C++11 lambda for providing reference values

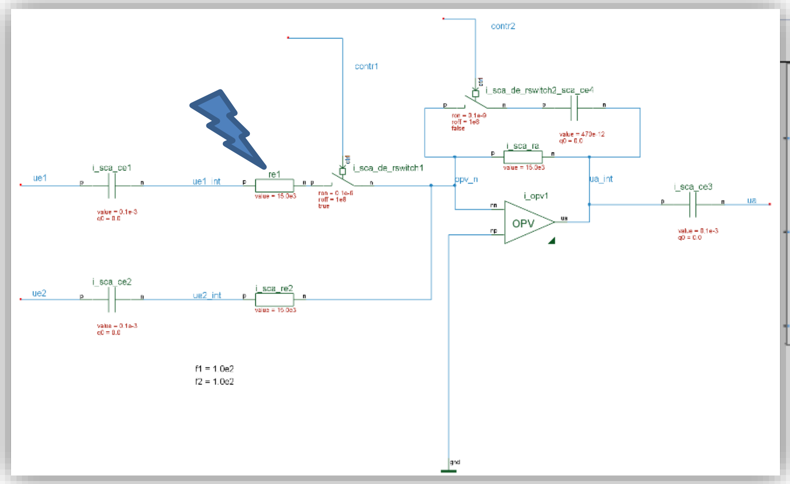
Application Example: OP-AMP with Fault Injection

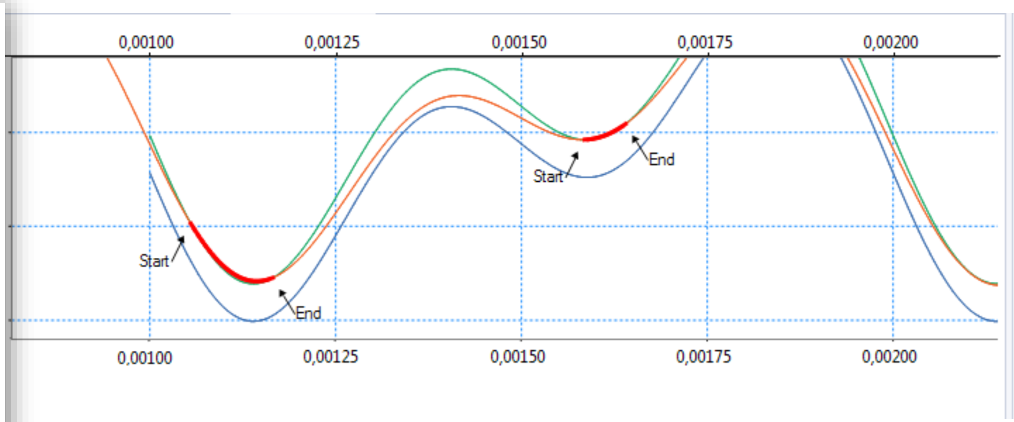


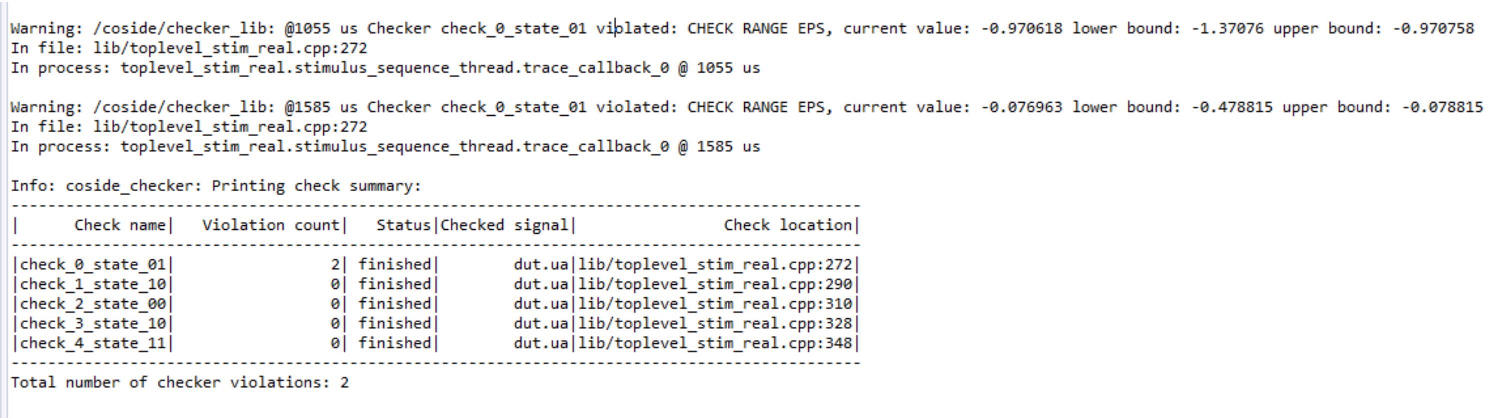


```
Info: coside_checker: Printing check summary:
------------------------------------------------------------------------------------------
|      Check name|Violation count|   Status|Checked signal|                Check location|
------------------------------------------------------------------------------------------
|check_0_state_01|              0| finished|        dut.ua|lib/toplevel_stim_real.cpp:272|
|check_1_state_10|              0| finished|        dut.ua|lib/toplevel_stim_real.cpp:290|
|check_2_state_00|              0| finished|        dut.ua|lib/toplevel_stim_real.cpp:310|
|check_3_state_10|              0| finished|        dut.ua|lib/toplevel_stim_real.cpp:328|
|check_4_state_11|              0| finished|        dut.ua|lib/toplevel_stim_real.cpp:348|
------------------------------------------------------------------------------------------
Total number of checker violations: 0
```

Application Example: OP-AMP with Fault Injection






```
Warning: /coside/checker_lib: @1055 us Checker check_0_state_01 violated: CHECK RANGE EPS, current value: -0.970618 lower bound: -1.37076 upper bound: -0.970758
In file: lib/toplevel_stim_real.cpp:272
In process: toplevel_stim_real.stimulus_sequence_thread.trace_callback_0 @ 1055 us

Warning: /coside/checker_lib: @1585 us Checker check_0_state_01 violated: CHECK RANGE EPS, current value: -0.076963 lower bound: -0.478815 upper bound: -0.078815
In file: lib/toplevel_stim_real.cpp:272
In process: toplevel_stim_real.stimulus_sequence_thread.trace_callback_0 @ 1585 us

Info: coside_checker: Printing check summary:
-----------------------------------------------------------------------------------------
|      Check name|   Violation count|   Status|Checked signal|              Check location|
-----------------------------------------------------------------------------------------
|check_0_state_01|                 2| finished|        dut.ua|lib/toplevel_stim_real.cpp:272|
|check_1_state_10|                 0| finished|        dut.ua|lib/toplevel_stim_real.cpp:290|
|check_2_state_00|                 0| finished|        dut.ua|lib/toplevel_stim_real.cpp:310|
|check_3_state_10|                 0| finished|        dut.ua|lib/toplevel_stim_real.cpp:328|
|check_4_state_11|                 0| finished|        dut.ua|lib/toplevel_stim_real.cpp:348|
-----------------------------------------------------------------------------------------
Total number of checker violations: 2
```

Summary

- Framework provides pre-defined checks for analog signal behavior.
- Allows to write checks independent of the SystemC AMS module of computation.
- Checks can be easily reused with sequences as they use relative timing.
- Easy creation of custom check algorithms within the framework.

Questions