2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Static Analysis of SystemC/SystemC-AMS System and Architectural Level Models

Karsten Einwich, Thilo Vörtler, COSEDA Technologies GmbH, Dresden, Germany
(karsten.einwich|thilo.voertler@coseda-tech.com)

***Abstract*—This paper introduces a generic concept and a framework for the static analysis of SystemC / SystemC AMS based system- and architectural level models based on SystemC sc_attributes. The framework permits the realization of analyzer for properties like power, area, resource utilization, gain, noise or IP2/IP3, whereby the properties can depend on the current system state.**

Keywords—SystemC, SystemC AMS, static analyzis, concept design, system design, architectural exploration

I. INTRODUCTION

In the concept-, system- and architectural-level design and exploration phases numerous parameters can be statically estimated. This can be general parameters like chip area estimation, power consumption, for digital systems resource utilization, in mixed signal systems this are important parameters like maximum output signal level, accumulated noise / noise figure, total harmonic distortion or RF design parameters like IP2/IP3 [1] (second and third order Interception point). In the simplest case the contributions of different subsystems are added up or calculated with explicit formulas. Conventionally, these parameters are analyzed using excel spread sheets, based on the elements of an RF chain. A static analysis is extremely fast compared to a dynamic analysis where the same parameters are extracted during simulation. With the increasing number of configuration modes and application scenarios, the maintenance of those spreadsheets becomes more and more complex. A second disadvantage is, that spread sheets are disconnected from the design, thus consistency must be established manually. Additionally, more complex topologies, e.g. multiple branches or loops are difficult to handle within spreadsheets.

To overcome the limitations of the excel approach, we introduce in this paper a framework, which permits to annotate and analyze attributes to modules and thus to subsystems of a SystemC/SystemC-AMS based system- or architectural level model. These attributes can be calculated in dependency of module parameters or the current state of the module, e.g. the current state of control signals. This allows an analysis for different configuration and operating states. As the analysis is based on the systems topology, also more complex structures e.g. consisting multiple branches can be handled.

The introduced framework utilizes the SystemC standard *sc_core::sc_attribute* class for the annotation of the static properties of subsystem components. The framework provides basic functionality, like attribute annotation and pre-analysis steps like setting up the dataflow/calculation graph. Based on this, specific analyses for different application domains can be realized easily. In this paper examples for performance analyzers and their implementation and usage will be presented.

II. GENERIC CONCEPTS

A SystemC/SystemC-AMS system- or architectural level model consists of modules (derived from *sc_module*), representing components or subsystems, which are connected via ports (derived from *sc_port*) and signals (derived from *sc_interface*). Ports can specify a direction (in or out). The introduced static analyzer is based on the IEEE P1666 SystemC standard [2] *sc_core::sc_attributes*. Thus, the static property is derived from this standard *sc_core::sc_attribut*e. A *sc_core::sc_attribute* consist of a name which is a string and a value, which is an object of an arbitrary type. The value object of the static analyzer base property provides base functionality like references to in and out ports as well as a method for calculating the value of the property. A static property can be attached to modules only. When a static property is attached to a *sc_module* (using the SystemC *add_attribute* method) per default the module ports are referenced to the attribute as in- or outports. Optionally, this referencing can be done manually to e.g. ignore certain ports for the analysis or determining the direction independent from the concrete port type. This assignment can be changed before an analysis is started. This allows to change the structure of the analyzed graph in dependency of the current state of the system (e.g. for multiplexer).

Static analysis can be based on different principles. E.g. for a power or chip area analysis the annotated values have to be added hierarchically. For a gain or noise analysis the values have to be calculated along the signal or data path. A third category are equation system based analyses, in this case the attribute contributes to an equation system, which is setup using structural information. The current framework implementation supports hierarchic and data flow based analyses, however it can be easily extended to support equation system based analysis too.

After annotation of the static properties, an analysis can be started. A concrete analyzer collects all attributes which belong to it by checking the attribute types through the hierarchy. In dependency of the analysis type, the attributes are analyzed with an abstract and thus generic dataflow or hierarchic analyzer. Thereby the value calculation functions of the attributes are executed. These functions calculate results in dependency of predecessor values (e.g. for data flow values from the referenced in ports) and the attribute contribution and propagate them to the successors.

Afterwards the results can be collected and printed to a shell or file. They can also be written into a format which allows the graphical annotation in schematic diagrams.

III. REALIZATION OF THE GENERIC CONCEPT

A. *Attributes*

Attributes are the hooks to annotate static properties to the model. Static analyzer attributes can be attached to modules (objects derived from *sc_core::sc_module*) only. All attributes belonging to a static analyzer must be derived from the common base class *cos_analyzer_attr_base* which themselves is derived from the SystemC *sc_core::sc_attribute* class. Thus, all SystemC mechanism like attaching attributes to objects and searching attributes are also available for static analyzer attributes.
Additionally, the *cos_analyzer_attr_base* class adds some basic functionality common for static analyzer attributes. These are functionalities like: attaching the attribute to a module, extracting and registration of the in- and outports of the module, providing a callback for the value function, accessing to predecessor values and propagating values to successors and access to properties provided by the generic analyzer like whether the attribute is inside the signal path. Figure 1 shows an excerpt of the static analyzer base class definition.

```
class cos_analyzer_attr_base : public sc_core::sc_attribute<cos_analyzer_value>
{
public:
    cos_analyzer_attr_base(sc_core::sc_module* parent);

    //access to number of predecessors / sucessors for a dataflow analysis
    unsigned long get_number_of_inports() const;
    unsigned long get_number_of_outports() const;

    template<class T>
    void set_outport_value(const T& val,long o); //predecessor/successor value access/ propagation

    template<class T>
    T get_inport_value(unsigned long i);

    template<class T>
    T get_module_value() const;

    //value callback function
    virtual void value_function(){}

     //access to dataflow analyzer result properties
    bool is_in_active_path();
    bool influences_active_path();
};
```

Figure 1: Excerpt of the static analyzer base class definition

B. *Analyzer*

All concrete analyzers will be derived from the analyzer base class *cos_static_analyzer_base* . This base class provides basic functionality like collecting all attributes, which belonging to a concrete analyzer and some generic utility functions for writing and formatting results. Additionally, it implements generic analyzers for different analyze classes. Currently a generic analyzer for a hierarchic analysis and a dataflow analyzer is

implemented. A third not yet realized class would be an equation system based analyzer. Figure 2 shows an excerpt of the analyzer base class.

```
class cos_static_analyzer_base
{
public:
    virtual void analyze() = 0; //callback to perform the analysis

    //start dataflow analysis
    void analyze_datapath(const std::vector<std::string>& starting_points);

    void analyze_hierarchy();  //start hierarchic analysis

    //set attribute exploration configuration
    void ignore_child_module_attributes();
    void dont_ignore_child_module_attributes();

    //utility functions for generic analysis result access
    sc_core::sc_object* get_toplevel_object();
    const vector<tree_analysis_result_element>& get_tree_analysis_result();
    const vector<std::vector<cos_analyzer_attr_base*> >& get_clusters();


    //callback function to recognize attributes which belong to
    //a concrete analyzer
    virtual bool is_type(cos_analyzer_attr_base*) = 0;
};
```

Figure 2: Excerpt of the static analyzer base class

1) Generic Hierarchic Analyzer

The hierarchic analyzer supports an analysis which combine attribute contribution of one hierarchy level and propagate them through the hierarchy. A typical example for a hierarchic analyzer is a power analyzer. In this case the power is annotated to e.g. primitive modules and simply the values of each hierarchy level are summed. The result will be a tree, whereby the root element contains the overall power and the branches represent the power contributions of the sub modules.
From a generic point of view, values of one hierarchy has to be combined – the current value has to be combined with a predecessor value and propagated as the predecessor for the next calculation. The final result will be attached as contribution of the module of the next hierarchy level.
This principle can be implemented as a generic concept, whereby the calculation (combination) of the current and predecessor value is implemented inside of the *value_function* callback of the concrete attribute (see Figure 1).

2) Generic Dataflow Analyzer

The dataflow analyzer supports an analysis which needs to combine attributes along a data- or signal path. A typical example is the analysis of the gain along a signal path. In this case the gain factors along a signal path have to be multiplied. Therefore, a classical dataflow analysis is performed – beginning at sources which are modules without inports, a scheduling/dependency graph is generated. This kind of analysis implies, that no loops are allowed. Afterwards, the *value_functions* of the attributes are executed in the order of the scheduling graph. Thus, the *value_functions* have to be combined the predecessor values from the inports with the own contribution and propagate them to the outports as predecessors for the next called value function.
Additionally, the analyzer permits to define a starting point. In this case the signal path from the specified starting point to end point(s), which are represented by modules without outports is analyzed. Modules which are inside this path – their calculation depends on the starting point module – are marked as inside the path. Modules which are not inside the signal path, however, at least one endpoint depends on their result are marked as have influence to the active path. Those properties are useful for analysis like a noise figure analysis, where an influence on the noise in the signal path is caused by modules from outside of the signal path.

C. Attribute Annotation

To enable a static analysis the attributes, have to be attached to modules. The first straightforward way is to annotate the property directly within the module description. Therefore, the attribute is instantiated inside the module context and will be automatically attached to the module using the SystemC method sc_add_attribute.

In other cases, a hierarchical composition of a static property is not easily possible. For such cases, the annotation of the static property to a hierarchical model is possible. In this case all properties of lower hierarchy (child) modules will be ignored (can be disabled) – thus we specify the contribution of the hierarchic module directly.
As in SystemC an object can hold only one attribute with the same name and an additional attachment will replace the attribute with the new one, this mechanism can be used to overrule a default attribute assignment (e.g. implemented inside the module) by an instance specific assignment e.g. done inside the netlist description or testbench.
The previously described annotation methods have the drawback, that the annotation is described within the specific SystemC modules. In a lot of cases such extensions are not possible or not convenient especially if legacy code is used. Therefore, a concept, which allows to implement a kind of shadow models was realized. This allows the description of a static attribute for a concrete module type. Using this concept, it is possible to automatically annotate the properties to all models of the corresponding types from the testbench or stimuli without any extension or change of the original model code. Technically, this is realized by traversing the hierarchy (using the SystemC *get_child_objects* methods) before starting an analysis and checking if a module type corresponds to the type of a shadow model. Is this the case the static property from the shadow model is attached to the module.
Summarized, three possibilities to annotate the attributes to the modules are supported:

1. Implementiaon inside the module constructor or one of the callbacks (e.g. *end_of_elaboration*)
2. Providing shadow models for certain module types
3. Attaching the module inside the netlist description or from the testbench

The order of the attachment allows a priority. Thus, attributes attached inside the module constructor can be overruled by shadow model definitions and this can be overruled by instance specific attributes assigned inside the netlist description or testbench. Additionally, the assignment of an attribute to a hierarchical module will overrule all attributes of the child modules. These mechanisms allow a very powerful control of the static analysis.

IV. Realization of Concrete Analyzers

A. *Hierarchic Analyzer Example – Power Analyzer*

This example demonstrates the realization of a simple power analyzer, which sums estimated power values hierarchically. Therefore, first a power attribute has to be implemented.

```
class cos_analyzer_attr_power : public cos_analyzer_attr_base
{
public:
          cos_analyzer_attr_power(sc_core::sc_module* mod, double val);

          void set_value(double val);
          double get_value() const;

          //datastructure to hold the values for the analysis
          struct cos_budget_value : public cos_value_base
          {
                    double value=0.0;
          } value;

          //call back function for calculating the power
          void value_function(cos_value_base&) override;

};

void cos_analyzer_attr_power::value_function(cos_value_base& previous)
{
       auto res=dynamic_cast<cos_power_value*>(&previous);
       //sum power values
       res->value+=this->get_value();
}
```

Figure 3: Excerpt of the attribute implementation of a power analyzer

Second the analyzer has to be implemented.

```
class cos_static_analyzer_power : public cos_static_analyzer_base
{
public:
      //perform analyzis
         void analyze() override;

       //store results in proprietary annotation format
         void store_annotations(const std::string& fname);
       //print results to shell
```

```
        void print_results(std::ostream& str=std::cout);
private:
       //callback for type recognition
        bool is_type(cos_analyzer_attr_base*) override;
};

void cos_static_analyzer_power::analyze()
{
   this->analyze_hierarchy(); //perform hierarchic analysis
}

bool cos_static_analyzer_power::is_type(cos_analyzer_attr_base* attr)
{
        return (dynamic_cast<cos_analyzer_attr_power*>(attr)!=NULL);
}

void cos_static_analyzer_power::print_results(std::ostream& str)
{
  auto& result=this->get_tree_analysis_result();

  double overall_power=result[0]->get_value<cos_analyzer_attr_power::cos_power_value>().value;

  str << "Overall power: " << overall_power << " W" << std::endl;

  for(auto& modp : result)
  {
     double mod_value=modp->get_value<cos_analyzer_attr_power::cos_power_value>().value;

     str << "\t" << modp->get_object()->name() << " :\t" << mod_value << " W" << std::endl;
  } }
```

Figure 4: Excerpt of the implementation of a power analyzer

If the attributes are attached to the modules, the analysis can be started at an arbitrary timepoint e.g. inside the sc_main function.

```
    cos_static_analyzer_power panalyzer;

    panalyzer.analyze();
    panalyzer.print_results();
    panalyzer.store_annotations("power_analysis_hierarchic");
```

Figure 5: Analyzer start and result printing

The result will look like shown in Figure 6.

```
Overall power: 4.31e-002W
        i_hierarchic_toplevel                 :       4.31e-002 W
        i_hierarchic_toplevel.i_cw_top_start_1 :      3.00e-003 W
        i_hierarchic_toplevel.i_cw_top_start_2 :      2.00e-003 W
        i_hierarchic_toplevel.i_gain_nl_gen1   :      5.00e-004 W
        i_hierarchic_toplevel.i_gain_nl_gen2   :      1.00e-003 W
        i_hierarchic_toplevel.i_muls_gen1      :      5.00e-004 W
        i_hierarchic_toplevel.i_sub_module1    :      3.30e-003 W
        i_hierarchic_toplevel.i_sub_module2    :      3.30e-003 W
        . . .
```

Figure 6: Analyzer result

Figure 7 shows a result visualization within the schematic editor of the COSIDE IDE.

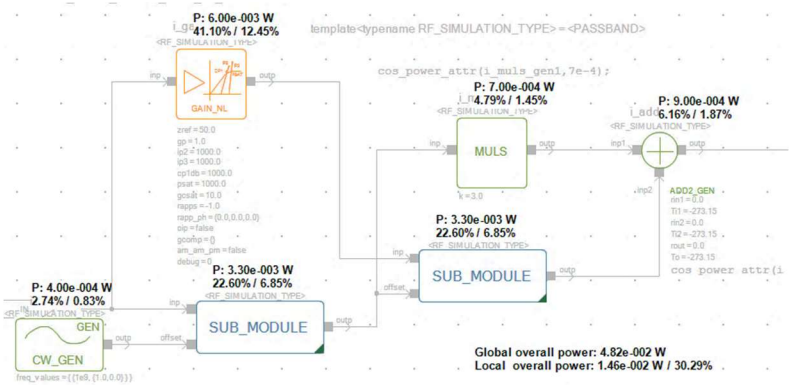

Figure 7: Result visualization

B. *Dataflow Analyzer Example – IIP2 / IIP3 Analyzer*

This example demonstrates the realization of a dataflow analyzer and shows also, that non-trivial analyses can be easily realized. IIP2 (second order input interception point) and IIP3 (third order input interception point) are measures for the non-linearity of RF systems. They are usually given as dBm values and related to a reference impedance (typical 50Ohm).

The resulting input third order interception point IIP3 of two cascaded amplifiers can be calculated by the following formula:

$$\frac{1}{IIP3^2} = \frac{1}{IIP3_1{}^2} + \frac{a_1{}^2}{IIP_2{}^2} \quad (1)$$

Whereby $IIP3_1$ is the input interception point of the first stage and $IIP3_2$ of the second stage and a_1 the gain of the first stage and IIP3 the resulting interception point – this formula uses linear gain and power values (not dB and dBm).

If two branches of the signal path are summed the resulting third order input interception point can be calculated by the following formula:

$$\frac{1}{IIP_2} = \left(\frac{a_1{}^2}{IIP3_1{}^2} + \frac{a_2{}^2}{IIP3_2{}^2}\right) \cdot \frac{1}{a_1 + a_2} \quad (2)$$

Similar (however different) formula existing for the IIP2. Due the gain values are required, the IIP2/IIP3 analysis implies also a gain analysis.

First, we have to provide the implementation of an attribute – an excerpt is shown in Figure 8.

```
class cos_rf_analyzer_ip2_ip3_attr : public cos_analyzer_attr_base
{
public:
        cos_rf_analyzer_ip2_ip3_attr(sc_core::sc_module* mod);

        void set_gain_db(double val,long i=-1,long o=-1); //attribute values setter
        void set_iip3_dbm(double value,double zref,long o=-1); …
private:
        struct cos_ip2ip3_value : public cos_value_base
        {
                double gain=0.0;
                double iip3_v_2=-1.0; //iip3 in volt**2
        };
        void value_function() override;
};

void cos_rf_analyzer_ip2_ip3_attr::value_function()
{
  for(unsigned long o=0;o<this->get_number_of_outports();++o)
  {
    double culm_gain= first_in_path?1.0:0.0;
    double old_gain=0.0, rez_sq_sum_iip3=0.0;

   //calculate sum of inports (formula (1) )
   for(unsigned long i=0;i<this->get_number_of_inports();++i)
   {
        double cgain= this->get_gain(i,o) * this->get_inport_value<cos_ip2ip3_value>(i).gain;
        culm_gain+=cgain;
        old_gain+=this->get_inport_value<cos_ip2ip3_value>(i).gain;

        //sum iip's of different paths
        double iip3=this->get_inport_value<cos_ip2ip3_value>(i).iip3_v_2;
        rez_sq_sum_iip3+=cgain/iip3;
   }
   ////////////// calculate cascade for IIP3 (formula (2) )///////////
   rez_sq_sum_iip3/=culm_gain;
   double i_iip3=1.0/rez_sq_sum_iip3;
   double iip3_term1=1.0 / i_iip3;
   double iip3_term2=old_gain*old_gain/ this->get_iip3_v_2();

   double iip3_term1_2=iip3_term1  + iip3_term2;
   double ciip3 = 1.0 /iip3_term1_2;

   cos_ip2ip3_value out_value;
   out_value.gain=culm_gain;
   out_value.iip3_v_2=ciip3;

   this->set_outport_value(out_value,o);
} } }
```

Figure 8: Excerpt of IIP3 attribute implementation

Second, the analyzer has to be implemented – Figure 9 shows the excerpt.

```
class cos_rf_analyzer_ip2_ip3 : public cos_static_analyzer_base
{
public:
         void analyze(const std::string& start_point);
         …
private:
         virtual bool is_type(cos_analyzer_attr_base*) override;
};

void cos_rf_analyzer_ip2_ip3::analyze(const std::string& start_point)
{
         this->analyze_datapath(start_point); }

bool cos_rf_analyzer_ip2_ip3::is_type(cos_analyzer_attr_base* attr)
{
         return (dynamic_cast<cos_rf_analyzer_ip2_ip3_attr*>(attr)!=NULL);
}
```

Figure 9: Excerpt of IIP3 analyzer implementation

After running the analyzer results can be visualized like shown in figure 10.

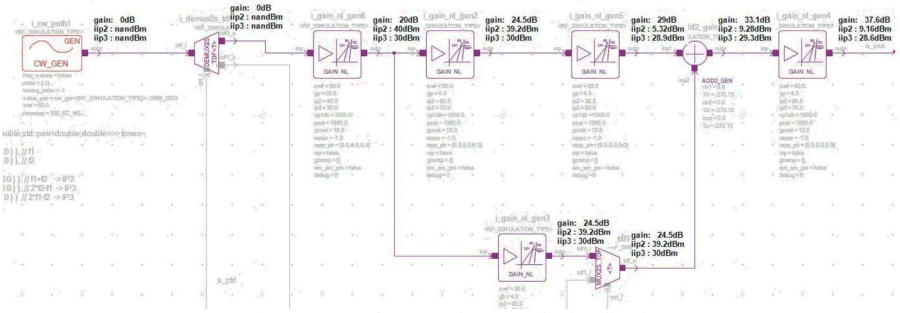

Figure 10: IIP2/IIP3 Analyzer result visualization

C. *Example for an Attribute Annotation which accesses a current module state*

An attribute can be implemented in a way that the value updates before executing an analysis, therefore an update callback can be provided. This allows to provide attributes which can change their value or there in- and outports in dependency of the system state. In Figure 11, this is illustrated on an implementation of a gain analyzer attribute.

```
class cos_rf_analyzer_gain_attr : public cos_analyzer_attr_base
{
public:
         cos_rf_analyzer_gain_attr(sc_core::sc_module* mod);
         void set_update_fct(std::function<void()> fct); //provide lambda function
         …
private: …
         std::function<void()> update_fct;
         void update() override;          //callback function executed before each analysis
         void value_function() override; //callback for value calculation
};

void cos_rf_analyzer_gain_attr::update()
{
   if(this->update_fct!=NULL) this->update_fct();
}
```

Figure 11: Using the update function to implement adoptable attributes

The defined update function can be provided if the attribute is attached to a module. Figure 12 illustrates this on an example of the implementation of a de-multiplexer, which directs the signalpath in dependency of a control inport.

```
SCA_CTOR(demux2s_tdf) //module constructor
{
  gattr=new cos_rf_analyzer_gain_attr(this);
  gattr->set_update_fct(  //define lambda function
     [&]() {
       if(mod->ctrl_i.read())  gattr->set_ports({&mod->tdf_i},{&mod->tdf1_o}); //set used in/outports
       else                    gattr.set_ports({&mod->tdf_i},{&mod->tdf0_o});  //in dependency of ctrl
       gattr->set_gain(1.0);});
}
```

Figure 12: Annotation of an attribute which accesses to the current module state

Figure 13 shows visualizations of results of a noise figure (a measure for the output noise power compared to the input noise power) analysis at different time points and thus with a different state of the control signal for the multiplexer/demultiplexer. Purple visualizes the current signal path, whereby the red module marks the selected starting point. Yellow modules are not in the signal path, however they produce noise which is added to the signal path and thus influence the noise figure. Light blue modules are also not in the signal path, however the noise they produce is not coupled into the signal path and thus has no influence to the current noise figure.

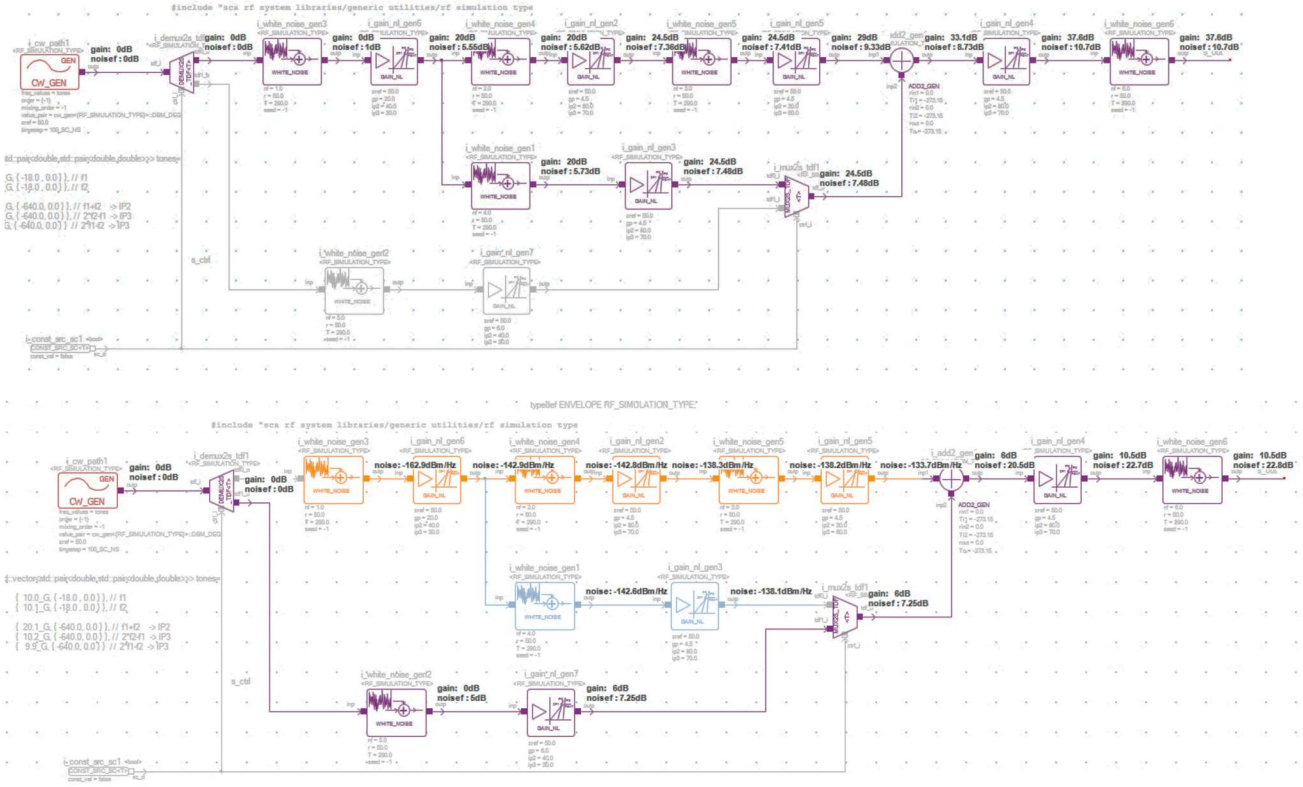

Figure 13: Visualization of the results of a static analysis of the noise figure for two different system states

V. SUMMARY

The paper presented a generic concept for static analysis based on the SystemC *sc_core::sc_attribute*. Based on the concept a framework was presented, which permits an easy implementation of different analyzers. The framework supports hierarchic analyzers, which can be used realize static analyzers for properties like power or chip area and data flow based analyzers which can be used to implement analyzers for properties like gain, noise or IP2/IP3. Based on this framework the implementation of a hierarchic analyzer and a data flow analyzer was presented. Additionally, it was shown, that the static analysis can be done in dependency of the current system state.

[1] Pozar, David M. “Microwave engineering”. John Wiley & Sons, 2009.

[2] “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

[3] “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