SystemC AMS Extensions

The Language

Karsten Einwich
Fraunhofer IIS/EAS Dresden
SystemC AMS 1.0 Standard released!

NEWS RELEASE

OSCI Completes First Analog/Mixed-Signal Standard for SystemC-based Design

SystemC AMS extensions enrich ESL design methodologies for mixed-signal applications

DATE CONFERENCE – DRESDEN, Germany – March 8, 2010 – The Open SystemC Initiative (OSCI), an independent non-profit organization dedicated to supporting and advancing SystemC™ as an industry-standard language for electronic system-level (ESL) design, today released the SystemC Analog/Mixed-signal (AMS) extensions language standard, AMS 1.0. The AMS 1.0 standard is the first modeling language targeting system-level design and verification to describe analog/mixed-signal behavior as natural extension to existing SystemC-based design methodologies.

As of 2005, the AMS language standardizes...
SystemC AMS extension

The standard package contains:

- Requirement specification document
- SystemC AMS extensions User’s Guide

Can be found on www.systemc.org

An open source (Apache 2) “proof-of-concept” implementation by Fraunhofer:
- http://systemc-ams.eas.iis.fraunhofer.de
OSCI AMS Working Group Roster

Steady growth in AMS WG: 53 individuals from 19 organizations
- strong drive from semiconductor industry
- full support of universities and research institutes
- growing interest and participation of EDA/ESL vendors

Chair: Martin Barnasconi, NXP Semiconductors
Vice chair: Christoph Grimm, Vienna University of Technology
SystemC AMS extensions 1.0

AMS methodology-specific elements
elements for AMS design refinement, etc.

Electrical Linear Networks (ELN)
modules
terminals
nodes

Linear Signal Flow (LSF)
modules
ports
signals

Timed Data Flow (TDF)
modules
ports
signals

Linear DAE solver

Scheduler

Synchronization layer

SystemC Language Standard (IEEE Std 1666-2005)

User features:
Classes and interfaces defined in AMS LRM Draft 1

Enabling technology:
Classes and interfaces not defined in AMS LRM Draft 1

Semantics defined in AMS LRM Draft 1

Source OSCI
SystemC-AMS is an extension of SystemC

- no changes to the SystemC implementation
- no restrictions for the use of the SystemC language
- use of the same SystemC implementation
- as far as possible, only LRM documented features used for the implementation of the library
Application Areas of SystemC-AMS

Modeling, Simulation and Verification for:

- Functional **complex** integrated systems (EAMS – Embedded Analogue Mixed Signal)
- Analogue **Mixed-Signal** systems / Heterogeneous systems
- **Specification** / Concept and System Engineering
- **System design**, development of a (“golden”) reference model
- Embedded **Software** development
- Next Layer (Driver) Software development
- **Customer model**, IP protection

- **not a replacement of Verilog/VHDL-AMS or Spice**
- **compared to Matlab, Ptolemy, … SystemC-AMS supports architectural exploration/refinement and software integration**
Outline

SystemC-AMS Language Composition
SystemC-AMS Model of Computation
  Timed Dataflow (TDF)
  Linear Signalflow (LSF)
  Electrical Linear Networks (ELN)
Types of analysis
  Time domain simulation
  Frequency / Noise domain simulation
Simulation Control and Debugging
  Testbenches / sc_main
  Tracing
  Postprocessing / Wave viewing
Proof-of-Concept Implementation
SystemC-AMS Language Composition
What’s different between analog and digital?

- Analog equation cannot be solved by the communication and synchronization of processes
  
  \[
  0 = i_t + \frac{v(w_{it})}{r_{it}} + c_{it} \frac{d(v(w_{it}) - v(w_{p}))}{dt}
  \]
  
  \[
  0 = \frac{v(w_{p})}{r_{prefi}} - c_{it} \frac{d(v(w_{it}) - v(w_{p}))}{dt}
  \]
  
  \[
  out = \frac{d}{dt}(k1 \cdot in + k2 \cdot out)
  \]
  
  -> in general an equation system must be setup

- The analog **system state changes continuously**
  
  - the value between solution points is continuous (linear is a first order approximation only)
  
  - -> the value of a time point between two solution points can be estimated only after the second point has been calculated (otherwise instable extrapolation)
Non Conservative vs. Conservative

Non Conservative

- Abstract representation of analog behavior
- The graph represents a continuous time (implicit) equation (system)

Conservative

- Represents topological structure of the modeled system
- Nodes are characterized by two quantities – the across value (e.g. voltage) and the through value (e.g. current)
- For electrical systems, Kirchhoff’s laws applied (KCL, KVL)
- For other physical domains generalized versions of Kirchhoffs’s laws applied
**SystemC-AMS Language Basics**

- A primitive **Module** represents a **contribution** of equations to a model of computation (MoC)
  - ->primitives of each MoC must be derived from a specific base class

- A **channel** represents in general an edge or variable of the equation system – thus not necessarily a communication channel

- SystemC-AMS modules/channels are **derived** from the SystemC base classes (**sc_module**, **sc_prim_channel/sc_interface**)

- There is no difference compared to SystemC for hierarchical descriptions – they are using **SC_MODULE / SCCTOR**
Symbol Names and Namespaces

- All SystemC-AMS symbols have the prefix `sca_` and macros the prefix `SCA_`

- All SystemC-AMS symbols are embedded in a namespace – the concept permits extensibility

- Symbols assigned to a certain MoC are in the corresponding namespace
  - `sca_tdf, sca_lsf, sca_eln`

- Symbols relating to core functionality or general base classes embedded in the namespace `sca_core`

- Symbols of utilities like tracing and datatypes are in the namespace `sca_util`

- Symbols related to small-signal frequency-domain analysis
  - `sca_ac_analysis`
SystemC-AMS Modules

- AMS modules are derived from `sca_core::sca_module` which is derived from `sc_core::sc_module`
  - Note: not all `sc_core::sc_module` member functions can be used

- AMS modules are always primitive modules
  - an AMS module can not contain other modules and/or channels
  - Hierarchical descriptions still use `sc_core::sc_module` (or `SC_MODULE` macro)

- Depending on the MoC, AMS modules are pre-defined or user-defined

- Language constructs
  - `sca_MoC::sca_module` (or `SCA_MoC_MODULE` macro)
  - e.g. `sca_tdf::sca_module` (or `SCA_TDF_MODULE` macro)
SystemC AMS channels

- AMS channels are derived from `sca_core::sca_interface` which is derived from `sc_core::sc_interface`
- AMS channels for Time Data Flow and Linear Signal Flow
  - based on directed connection
  - used for non-conservative AMS model of computation
  - Language constructs:
    - `sca_MoC::sca_signal`
    - e.g. `sca_lsf::sca_signal`, `sca_tdf::sca_signal<T>`
- AMS channels for Electrical Linear Networks
  - conservative, non-directed connection
  - characterized by an across (voltage) and through (current) value
  - Language constructs:
    - `sca_MoC::sca_node / sca_MoC::sca_node_ref`
    - e.g. `sca_eln::sca_node`, `sca_eln::sca_node_ref`
SystemC AMS Language Composition - Summarize

- **sca_module** – base class for SystemC AMS primitive
- **sca_in / sca_out** – non-conservative (directed in/outport)
- **sca_terminal** – conservative terminal
- **sca_signal** – non-conservative (directed) signal
- **sca_node / sca_node_ref** – conservative node

The MoC is assigned by the namespace e.g.:

- **sca_tdf::sca_module** - base class for timed dataflow primitives modules
- **sca_lsf::sca_in** - a linear signalflow inport
- **sca_tdf::sca_in<int>** - a TDF inport
- **sca_eln::sca_terminal** - an electrical linear network terminal
- **sca_eln::sca_node** - an electrical linear network node
SystemC AMS Language Element Composition - Converter

Converter elements are composed by the namespaces of both domains:

- **sca_tdf::sc_de::sca_in<T>** - is a port of a TDF primitive module, which can be connected to an `sc_core::sc_signal<T>` or to a `sc_core::sc_in<T>`
  - Abbreviation: `sca_tdf::sc_in<T>`

- **sca_eln::sca_tdf::sca_voltage** – is a voltage source which is controlled by a TDF input
  - Abbreviation: `sca_eln::sca_tdf_voltage`

- **sca_lsf::sc_core::sca_source** – is a linear signal flow source controlled by a SystemC signal (`sc_core::sc_signal<double>`)
  - Abbreviation: `sca_lsf::sca_sc_source`
Include systemc-ams versus systemc-ams.h

- **systemc-ams** includes systemc and all SystemC-AMS class, symbol and macro definitions

- **systemc-ams.h** includes systemc-ams and systemc.h and adds all symbols of the following namespaces to the global namespace (by e.g. use `sca_util::sca_complex;`):
  - sca_ac_analysis
  - sca_core
  - sca_util

- **Note:** Symbols of MoC related namespaces are not added
Timed Dataflow - TDF
**Dataflow Basics**

- Simple firing rule: A module is executed if enough samples available at its input ports.

- The function of a module is performed by:
  - reading from the input ports (thus consuming samples),
  - processing the calculations and
  - writing the results to the output ports.

- For synchronous dataflow (SDF) the numbers of read/written samples are constant for each module activation.

- The scheduling order follows the signal flow direction.

**Equation system:**

\[
\begin{align*}
    s_1 &= \text{in} \\
    s_2 &= f_1(s_1) \\
    \text{out} &= f_2(s_2)
\end{align*}
\]
Loops in Dataflow Graphs

- Graphs with loops require a delay to become schedulable

- A delay inserts a sample in the initialization phase
Multi Rate Dataflow Graphs

- The number of read/write sample (rate) is for at least one port >1 -> multi rate
- The rates in loops must be consistent
Timed Dataflow

- Dataflow is an untimed MoC
- Timed dataflow tags each sample and each module execution with an absolute time point
- Therefore the time distance (timestep) between two sample/two executions is assumed as constant
- This time distance has to be specified
- Enables synchronization with time driven MoC like SystemC discrete event and embedding of time dependent functions like a continuous time transfer function
TDF – Timestep Propagation

- If more than one timestep assigned consistency will be checked
TDF Attributes - Summarize

- **rate**
  - Port attribute – number of sample for reading / writing during one module execution

- **delay**
  - Port attribute – number of sample delay, number of samples to be inserted while initializing

- **timestep**
  - Port and module attribute – time distance between two samples or two module activations
Synchronization between TDF and SystemC DE

- Synchronization between SystemC discrete event (DE) is done by converter ports.

- They have the same attributes and access methods like usual TDF ports.

- SystemC (DE) signals are sampled at the first $\Delta$ of the tagged TDF time point.

- TDF samples are scheduled at the first $\Delta$ of the tagged TDF time (and thus valid at least at $\Delta=1$).
TDF Elaboration and Simulation

**TDF elaboration phase**

- **TDF module attribute settings:**
  - Execute all `set_attributes` methods

- **TDF time step calculation and propagation:**
  - Define time step and check consistency

- **TDF cluster computability check:**
  - Define and check cluster schedule

- **TDF module initialization:**
  - Execute all `initialize` methods once

**TDF simulation phase**

- **TDF module activation and processing:**
  - Continuously execute all `processing` methods

- **TDF module post processing:**
  - Execute all `end_of_simulation` methods once
Summarize TDF MoC

cluster = set of connected TDF modules

Port timestep = 1/16e3

16 kHz

ipl

2

A

1

2

dec

1

8 kHz

delay

1

1

B

1

ctrl

port rates

call order of modules

ipl

A

A

dec

B

del

Scheduling List

setup SystemC DE synchronization

embedding in SystemC process

SystemC process executed every cluster timestep

Block Diagram

sampling rate calculation / check

timestep calculation / check

scheduling list creation / calculation cluster timestep

SystemC module

convertor port

sca_tdf::sca_signal

sca_tdf::sca_signal

Grenoble Tutorial 7.9.2010

© Fraunhofer IIS/EAS
Timed Dataflow (TDF) Primitive Module

- Module declaration macros
  
  ```cpp
  SCA_TDF_MODULE(<name>)
  struct <name>:
    public sca_tdf::sca_module
  ```

- Port declarations dataflow ports
  
  ```cpp
  sca_tdf::sca_in< <T> >,
  sca_tdf::sca_out< <T> >
  ```

- Port declaration converter ports
  (for TDF primitives only)
  
  ```cpp
  sca_tdf::sc_in< <T> >,
  sca_tdf::sc_out< <T> >
  ```

- Virtual primitive methods called by the simulation kernel – overloaded by the user defined tdf primitive
  
  ```cpp
  void set_attributes()
  void initialize()
  void processing()
  void ac_processing()
  ```

- Methods for set/get module activation timestep
  
  ```cpp
  void set_timestep(const sca_time&);
  sca_time get_time()
  ```

- Constructor macro / constructor
  
  ```cpp
  SCA_CTOR(<name>)
  <name>(sc_module_name nm)
  ```
Structure Timed Dataflow User defined Primitive

```cpp
SCA_TDF_MODULE(mytdfmodel)    // create your own TDF primitive module
{
    sca_tdf::sca_in<double>  in1, in2;  // TDF input ports
    sca_tdf::sca_out<double> out;       // TDF output port

    void set_attributes()
    {   // placeholder for simulation attributes
        // e.g. rate: in1.set_rate(2); or delay: in1.set_delay(1);
    }

    void initialize()
    {   // put your initial values here e.g. in1.initialize(0.0);
    }

    void processing()
    {   // put your signal processing or algorithm here
    }

    SCA_CTOR(mytdfmodel) {}  
};
```
Set and get TDF Port Attributes

- Set methods can only be called in `set_attributes()`
- Get methods can be called in `initialize()` and `processing()`

- Sets / gets port rate (number of samples read/write per execution)
  - void `set_rate(unsigned long rate)`
  - unsigned long `get_rate()`

- Set/get number of sample delay
  - void `set_delay(unsigned long nsamples)`
  - unsigned long `get_delay()`

- Set time distance of samples get calculated/propagated time distance
  - void `set_timestep(const sca_time&)`
  - sca_time `get_time_step()`

- Get absolute sample time
  - sca_time `get_time(unsigned long sample)`
TDF Port read and write Methods

- Writes initial value to delay buffer
  - only allowed in `initialize()`
  - `sample_id` must be smaller than the number of delays
  - available for all in- and outports

- Reads value from inport
  - only allowed in `processing()`
  - `sca_tdf::sca_in<T>` or `sca_tdf::sca_de::sca_in<T>`

- Writes value to outport
  - only allowed in `processing()`
  - `sca_tdf::sca_out<T>` or `sca_tdf::sca_de::sca_out<T>`

```cpp
void initialize(
    const T& value,
    unsigned long sample_id=0)

const T& read(
    unsigned long sample_id=0)

operator const T&() const
const T& operator[]
    (unsigned long sample_id) const

void write( const T& value,
    unsigned long sample_id=0)
...
operator= (const T&)
...
operator[](unsigned long sample_id)
```
First complete TDF Primitive Module

```cpp
SCA_TDF_MODULE(mixer) // TDF primitive module definition
{
   sca_tdf::sca_in<double> rf_in, lo_in; // TDF in ports
   sca_tdf::sca_out<double> if_out;     // TDF out ports

   void set_attributes()
   {
      set_timestep(1.0, SC_US); // time between activations
      if_out.set_delay(5);     // 5 sample delay at port if_out
   }

   void initialize()
   {
      // initialize delay buffer (first 5 sample read by the
      // following connected module inport)
      for(unsigned int i=0; i<5; i++) if_out.initialize(0.0,i);
   }

   void processing()
   {
      if_out.write( rf_in.read() * lo_in.read() );
   }

   SCA_CTOR(mixer)
};
```
Linear Dynamic Behavior for TDF Models

- TDF Models can embed linear equation systems provided in the following three forms:

\[ H(s) = \frac{b_n \cdot s^n + b_{n-1} \cdot s^{n-1} + \ldots + b_0}{a_m \cdot s^m + a_{m-1} \cdot s^{m-1} + \ldots + a_0} \]

- Linear transfer function in numerator / denominator representation

\[ H(s) = k \cdot \frac{(s-z_0) \cdot (s-z_1) \cdot \ldots \cdot (s-z_n)}{(s-p_0) \cdot (s-p_1) \cdot \ldots \cdot (s-p_n)} \]

- Linear transfer function in pole-zero representation

\[ \dot{x} = Ax + Bu \]
\[ y = Cx + Du \]

- State Space equations
Linear Dynamic Behavior for TDF Models  2/2

- The equation systems will be represented and calculated by objects:
  - `sca_tdf::sca_ltf_nd` - Numerator / denumerator representation
  - `sca_tdf::sca_ltf_zp` - Pole-zero representation
  - `sca_tdf::sca_ss` - State space equations

- The result is a continuous time signal represented by a “artificial” object (`sca_tdf::sca_ct_proxy` or `sca_tdf::sca_ct_vector_proxy`)
  - This object performs the time discretization (sampling) in dependency of the context – this makes the usage more comfortable and increases the accuracy
  - This mechanism permits additionally a very fast calculation for multi-rate systems
**TDF Module – Example with LTF**

```cpp
SCA_TDF_MODULE(prefi_ac) {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;

    //control / DE signal from SystemC
    //((connected to sc_signal<bool>))
    sca_tdf::sc_in<bool> fc_high;

double fc0, fc1, v_max;

    //filter equation objects
    sca_tdf::sca_ltf_nd ltf_0, ltf_1;
    sca_util::sca_vector<double> a0, a1, b;
    sca_util::sca_vector<double> s;

    void initialize() {
        const double r2pi = M_1_PI * 0.5;
        b(0) = 1.0; a1(0)=a0(0)= 1.0;
        a1(1)= r2pi/fc0; a1(1) = r2pi/fc1;
    }

    void processing() {
        double tmp;
        if(fc_high) tmp = ltf_1(b,a1,s,in);
        else tmp = ltf_0(b,a0,s,in);

        //output value limitation
        if (tmp > v_max) tmp = v_max;
        else if (tmp < -v_max) tmp = -v_max;

        out.write(tmp);
    }

    SCA_CTOR(prefi_ac) {
        //default parameter values
        fc0 = 1.0e3; fc1=1.0e5; v_max=1.0;
    }
};
```

\[
H(s) = \frac{1}{1 + \frac{1}{2\pi f_c} s}
\]
Hierarchical Module Example

```cpp
SC_MODULE(hier) {
    sca_tdf::sca_in<double>  in;
    sca_tdf::sca_out<double> out;
    sc_in<bool>              lpx_on;
    delay*  delay_i;
    lpx*    lpx_i;
    sca_tdf::sca_signal<double> lpx2delay;

    SC_CTOR(hier) {
        lpx_i = new lpx("lpx_i");
        lpx_i->in(in);
        lpx_i->out(lpx2delay);
        lpx_i->on(lpx_on);
        delay_i = new delay("delay_i");
        delay_i->in(lpx2delay);
        delay_i->out(out);
        delay_i->delay_val = 5;
    }
};
```
Hierarchical Composition – Binding rules

- Child `sca_tdf::sca_in<T>/sca_out<T>` to `sca_tdf::sca_signal<T>`
- Child `sca_tdf::sca_in<T>` to parent `sca_tdf::sca_in<T>`
- Child `sca_tdf::sca_out<T>` to parent `sca_tdf::sca_out<T>`
- Child `sca_tdf::sca_in<T>` to parent `sca_tdf::sca_out<T>`

- Primitive `sca_tdf::sc_in<T>/sca_tdf::sc_out<T>` to `sc_signal<T>`
- Primitive `sca_tdf::sc_in<T>` to parent `sc_in<T>`
- Primitive `sca_tdf::sc_out<T>` to parent `sc_out<T>`
- Primitive `sca_tdf::sc_in<T>` to parent `sc_out<T>`

- Always: exactly one primitive outport to an arbitrary number of primitive inports throughout the hierarchy (each primitive inport must be connected to exactly one primitive outport)

- Not possible: Parent inport to parent outport -> Dummy module required
TDF Model Composition

SC_MODULE

TDF input port
sca_tdf::sca_in<T>

Port-to-port binding

in_tdf

Discrete-event input port
sc_core::sc_in<T>

TDF input converter port
sca_tdf::sca_de::sca_in<T>

Port-to-port binding

in_sc

TDF

Out Sc

C

in1

in2

in3

out1

out2

Port-to-port binding

TDF signal
sca_tdf::sca_signal<T>

out_tdf

out_sc

SystemC DE

Discrete-event signal
sc_core::sca_signal<T>

TDF input port
sca_tdf::sca_out<T>

Port-to-port binding

out1

out2

TDF output converter port
sca_tdf::sca_de::sca_out<T>

Discrete-event output port
sc_core::sc_out<T>
Hierarchical Module

```cpp
SC_MODULE(my_hierarchical) {
    sca_tdf::sca_in<int> in_tdf;
    sca_tdf::sca_out<double> out_tdf;
    sc_in<double> in_sc;
    sc_out<bool> out_sc;
    sc_signal<double> sc_sig1;
    sc_signal<sc_logic> sc_sig2;
    sca_tdf::sca_signal<bool> tdf_sig;
    module_tdf_c* tdf_c;
    module_tdf_d* tdf_d;
    module_sc_e* e_sc;

    SC_CTOR(my_hierarchical) {
        tdf_c=new module_tdf_c("tdf_c");
        tdf_c->in1(in_sc);
        tdf_c->in2(in_tdf);
        tdf_c->in3(out_tdf);
        tdf_c->out1(sc_sig1);
        tdf_c->out2(tdf_sig);
        tdf_d=new module_tdf_d("tdf_d");
        tdf_d->in1(tdf_sig);
        tdf_d->in2(sc_sig2);
        tdf_d->out1(out_sc);
        tdf_d->out2(out_tdf);
        e_sc = new module_sc_e("e_sc");
        e_sc->in(sc_sig1);
        e_sc->out(sc_sig2);
    }
};
```
Linear Signalflow (LSF) Modeling

- Library of predefined elements
- Permits the description of arbitrary linear equation systems
- Several converter modules to/from TDF and SystemC \((\text{sc}\_\text{core}::\text{sc}\_\text{signal})\)
- Models for switching behavior like mux / demux

- LSF models are always hierarchical models

- Ports:
  - \texttt{sca\_lsf::sca\_in} - input port
  - \texttt{sca\_lsf::sca\_out} - output port

- Channel / Signal:
  - \texttt{sca\_lsf::sca\_signal}
LSF predefined modules

- `sca_lsf::sca_add`
- `sca_lsf::sca_sub`
- `sca_lsf::sca_gain`
- `sca_lsf::sca_dot`
- `sca_lsf::sca_integ`
- `sca_lsf::sca_delay`
- `sca_lsf::sca_source`
- `sca_lsf::sca_ltf_nd`
- `sca_lsf::sca_ltf_zp`
- `sca_lsf::sca_ss`
- `sca_lsf::sca_tdf::sca_source`
- `sca_lsf::sca_tdf::sca_gain`
- `sca_lsf::sca_tdf::sca_mux`
- `sca_lsf::sca_tdf::sca_demux`
- `sca_lsf::sca_tdf::sca_sink`
- `sca_lsf::sc_de::sca_source`
- `sca_lsf::sc_de::sca_gain`
- `sca_lsf::sc_de::sca_mux`
- `sca_lsf::sc_de::sca_demux`
- `sca_lsf::sc_de::sca_sink`
Example: LSF language constructs

```cpp
SC_MODULE(mylsfmodel) // create a model using LSF primitive modules {
    sca_lsf::sca_in in;  // LSF input port
    sca_lsf::sca_out out;  // LSF output port
    sca_lsf::sca_signal sig;  // LSF signal
    sca_lsf::sca_dot* dot1;  // declare module instances
    sca_lsf::sca_sub* sub1;

    mylsfmodel(sc_module_name, double fc=1.0e3)
    {
        // instantiate predefined primitives
        dot1 = new sca_lsf::sca_dot("dot1", 1.0/(2.0*M_PI*fc));
        dot1->x(out);
        dot1->y(sig); // parameters

        sub1 = new sca_lsf::sca_sub("sub1");
        sub1->x1(in);
        sub1->x2(sig);
        sub1->y(out);
    }
};
```
Hierarchical Composition – Binding rules

- Child `sca_lsf::sca_in / sca_out` to `sca_lsf::sca_signal`
- Child `sca_lsf::sca_in` to parent `sca_lsf::sca_in`
- Child `sca_lsf::sca_out` to parent `sca_lsf::sca_out`
- Child `sca_lsf::sca_in` to parent `sca_lsf::sca_out`

- Exactly one `sca_lsf::sca_out` to an arbitrary `sca_lsf::sca_in` throughout the hierarchy (each `sca_lsf::sca_in` must be connected to exactly one primitive `sca_lsf::sca_out` via a `sca_lsf::sca_signal`)

- Not possible: Parent inport to parent outport -> Dummy e.g. `sca_lsf::sca_gain` module required
Electrical Linear Networks - ELN
Electrical Linear Network (ELN) Modeling

- Library of predefined elements
- Permits the description of arbitrary linear electrical network
- Several converter modules to/from TDF and SystemC (*sc_core::sc_signal*)
- Models for switching behavior like switches

- ELN models are always hierarchical models

- Ports:
  - *sca_eln::sca_terminal* - conservative terminal

- Channel / Node:
  - *sca_eln::sca_node* – conservative node
  - *sca_eln::sca_node_ref* – reference node, node voltage is always zero
ELN predefined elements

- `sca_eln::sca_r`
- `sca_eln::sca_l`
- `sca_eln::sca_c`
- `sca_eln::sca_v cvs`
- `sca_eln::sca_v ccs`
- `sca_eln::sca_c cv s`
- `sca_eln::sca_c c cs`
- `sca_eln::sca_null or`
- `sca_eln::sca_g y r at or`
- `sca_eln::sca_ideal_transformer`
- `sca_eln::sca_transmission_line`
- `sca_eln::sca_vsource`
- `sca_eln::isource`

- `sca_eln::sca_tdf::sca_vsink`
- `sca_eln::sca_tdf_vsink`
- `sca_eln::sca_tdf::sca_vsource`
- `sca_eln::sca_tdf_vsource`
- `sca_eln::sca_tdf::sca_isource`
- `sca_eln::sca_tdf_isource`
- `sca_eln::sc_de::sca_vsource`
- `sca_eln::de_vsource`
- `sca_eln::sc_de::sca_isource`
- `sca_eln::sc_de::sca_isource ...
- `sca_eln::sca_tdf::sca_r ...
- `sca_eln::sca_tdf::sca_l ...
- `sca_eln::sca_tdf::sca_c ...
- `sca_eln::sc_de::sca_r ...
- `sca_eln::sc_de::sca_l ...
- `sca_eln::sc_de::sca_c ...
- `...

Grenoble Tutorial 7.9.2010
Example: ELN language constructs

```cpp
SC_MODULE(myelnmodel) // model using ELN primitive modules {
    sca_eln::sca_terminal in, out; // ELN terminal (input and output)

    sca_eln::sca_node n1; // ELN node
    sca_eln::sca_node_ref gnd; // ELN reference node

    sca_eln::sca_r *r1, *r2;
    sca_eln::sca_c *c1;

    SC_CTOR(myelnmodel) // standard constructor {
        r1 = new sca_eln::sca_r("r1"); // instantiate predefined
        r1->p(in); // primitive here (resistor)
        r1->n(out);
        r1->value = 10e3; //named parameter association

        c1 = new sca_eln::sca_c("c1", 100e-6); //positional parameter association
        c1->p(out);
        c1->n(n1);

        r2 = new sca_eln::sca_r("r2",100.0);
        r2->p(n1);
        r2->n(gnd);
    }
}
```

Diagram:
- A circuit with two resistors (r1 and r2) and a capacitor (c1) connected to terminals in, out, and gnd.
- r1 is a 10kΩ resistor,
- c1 is a 100uF capacitor.
Solvability of Analog Equations (also for LSF)

- Not all analog systems which can be described are solvable
- Not all theoretically solvable analog systems can be solved by the applied numerical algorithm
- Do not connect voltage sources in parallel or current sources in series
- Do resistor with the value zero representing a short cut (voltage source with value zero)
- Do not have floating nodes
- You need always a path to a reference node
- The time constants of the network should be at least ~5 times larger than the simulation step width
- Prevent the use of extremely high/low values and large differences in the dimensions
Types of Analysis
Analysis Types

- Transient time domain is driven by the SystemC kernel
  - thus the SystemC `sc_core::sc_start` command controls the simulation

- Two different kinds of small-signal frequency-domain analysis (AC analysis) are available
  - AC-analysis
  - AC-noise-analysis
Small Signal Frequency Domain Analysis (AC-Analysis)

- AC-analysis:
  - Calculates linear complex equation system stimulated by AC-sources

- AC noise domain
  - solves the linear complex equation system for each noise source contribution (other source contributions will be neglected)
  - adds the results arithmetically

- ELN and LSF description are specified in the frequency domain

- TDF description must specify the linear complex transfer function of the module inside the method `ac_processing` (otherwise the out values assumed as zero)

- This transfer function can depend on the current time domain state (e.g. the setting of a control signal)
Small-Signal Frequency-Domain Analysis

Linear equation system contribution for LSF/ELN:

$$q(t) = Ax + Bx \rightarrow q(f) = Ajwx(f) + Bx(f)$$

Sources
Frequency Domain Description for TDF Models

SCA_TDF_MODULE(combfilter) {
    sca_tdf::sca_in<bool> in;
    sca_tdf::sca_out<sc_int<28>> out;

    void set_attributes() {
        in.set_rate(64); // 16 MHz
        out.set_rate(1); // 256 kHz
    }

    void ac_processing() {
        double k = 64.0;
        double n = 3.0;

        // complex transfer function:
        sca_complex h;
        h = pow((1.0 - sca_ac_z(-k)) / (1.0 - sca_ac_z(-1)), n);

        sca_ac(out) = h * sca_ac(in);
    }

    void processing() {
        int x, y, i;
        for (i=0; i<64; ++i) {
            x = in.read(i);
            ...out.write(y);
        }

        SCACTOR(combfilter) {
            ...
        }
    }
};

\[
H(z) = \left(\frac{1-z^{-k}}{1-z^{-1}}\right)^n
\]

\[
z = e^{j2\pi f/\nu_s}
\]
Simulation Control and Debugging
Tracing of analog Signals

- SystemC AMS has a own trace mechanism:
  - Analog / Digital timescales are not always synchronized
  - Note: the VCD file format is in general inefficient for analog

- Traceable are:
  - all `sca_<moc>::sca_signal`'s, `sca_eln::sca_node` (voltage) and `sc_core::sc_signal`
  - Most ELN modules – the current through the module
  - ports and terminals (traces the connected node or signal)
  - for TDF a traceable variable to trace internal model states

Two formats supported:
- Tabular trace file format
  - `sca_util::sca_create_tabular_trace_file`
- VCD trace file format
  - `sca_util::sca_create_vcd_trace_file`

Features to reduce amount of trace data:
- enable / disable tracing for certain time periods, redirect to different files
- different trace modes like: sampling / decimation
Viewing Wave Files

- Simple Tabular Format:

  ```plaintext
  %time name1 name2  ...
  0.0  1     2.1  ...
  0.1  1e2   0.3  ...
  :     :     :    :  
  ```

- A lot of tools like gwave or gaw can read this format

- Can be load directly into Matlab/Octave by the load command:
  ```matlab
  load result.dat
  plot(result(:,1), result(:,2)); %plot the first trace versus time
  plot(result(:,1), result(:,2:end)); %plot all waves versus time
  ```

- For compatibility with SystemC the vcd Format is available
  - however it is not well suited to store analogue waves
  - VCD waveform viewers usually handle analogue waves badly
Simulation Control

- Time domain – no difference to SystemC
  - `sc_start(10.0,SC_MS);`  // run simulation for 10 ms
  - `sc_start();`  // run simulation forever or `sc_stop()` is called

- AC-domain / AC-noise-domain
  - Run simulation from 1Hz to 100kHz, calculate 1000 points logarithmically spaced:
    - `sca_ac_start(1.0,100e3,1000,SCA_LOG);`  // ac-domain
    - `sca_ac_noise_start(1.0,100e3,1000,SCA_LOG);`  // ac-noise domain
  - Run simulation at frequency points given by a `std::vector<double>`:
    - `sca_ac_start(frequencies);`  // ac-domain
    - `sca_ac_noise_start(frequencies);`  // ac-noise domain
SystemC AMS Testbench 1/2

#include <systemc-ams.h>

int sc_main(int argn, char* argc)
{
    // instantiate signals, modules, ... from arbitrary domains e.g.:
    sca_tdf::sca_signal<double> s1;
    sca_eln::sca_node n1;
    sca_lsf::sca_signal slsf1;
    sca_core::sca_signal<bool> scsig1;

    dut i_dut("i_dut");
    i_dut->inp(s1);
    u_dut->ctrl(scsig1);

    sc_trace_file* sctf=sc_create_vcd_trace_file("sctr");
    sc_trace(sctf,scsig1,"scsig1"); ...

SystemC AMS Testbench 2/2

```c
sca_trace_file* satf=sca_create_tabular_trace_file("tr.dat");
sca_trace(satf,n1,"n1"); …

sc_start(2.0,SC_MS); // start time domain simulation for 2ms
satf->disable(); // stop writing
sc_start(2.0,SC_MS); // continue 2ms
satf->enable(); // continue writing
sc_start(2.0,SC_MS); // continue 2ms

// close time domain file, open ac-file
satf->reopen("my_tr_ac.dat");

sca_ac_start(1.0,1e6,1000,SCA_LOG); // calculate ac at current op
// reopen transient file, append
satf->reopen("mytr.dat",std::ios::app);

// sample results with 1us time distance
satf->set_mode(sca_sampling(1.0,SC_US));
sc_start(100.0,SC_MS); // continue time domain

sc_close_vcd_trace_file(sctf); // close SystemC vcd trace file
sca_close_tabular_trace_file(satf); // close tabular trace file
```


Proof-of-Concept Implementation
Getting Started

1. Download SystemC from [www.systemc.org](http://www.systemc.org) and the SystemC-AMS proof-of-concept from [http://systemc-ams.eas.iis.fraunhofer.de](http://systemc-ams.eas.iis.fraunhofer.de)

2. Install the libraries for Linux, Solaris, Windows/MinGW or Windows/cygwin (preferable 32 Bit) , and preferable gcc > 4.x required (read readme for details)
   - tar –xzvf systemc.tar.gz
   - cd systemc
   - configure
   - make; make install
   - setenv SYSTEMC_PATH <your install dir> -> may insert in your .cshrc

   - tar –xzvf systemc_ams.tar.gz
   - cd systemc_ams
   - configure
   - make; make install
   - setenv SYSTEMC_AMS_PATH <your install dir> -> may insert in your .cshrc
Getting Started SystemC Files

Recommendations:
- Split the module description into a header and a cpp implementation file
- Only one module per header / cpp file
- The name of the module shall be equal to the header / cpp file name
- Do not use capital letters and special characters (like ä, %, &, space, …)
SystemC / SystemC-AMS Compilation

```
g++ -c module1.cpp -I${SYSTEMC_PATH}/include \  
   -I${SYSTEMC_AMS_PATH}/include

g++ -c module2.cpp -I${SYSTEMC_PATH}/include \  
   -I${SYSTEMC_AMS_PATH}/include

ar -rcs libmodels.a module1.o module2.o

```

```
g++ -c tb.cpp -I${SYSTEMC_PATH}/include \  
   -I${SYSTEMC_AMS_PATH}/include

```

```
g++ -c tb.cpp -L${SYSTEMC_PATH}/lib-{TARGET_ARCH} \  
   -L${SYSTEMC_AMS_PATH}/lib-{TARGET_ARCH}  
   -L. -o tb.exe -lmodels -lsystemc-ams -lsystemc

``` !!!! Library order Important !!!!
More …

- www.systemc.org
- www.systemc-ams.org
- www.systemc-ams.eas.iis.fraunhofer.de