SystemC-AMS
Analog & Mixed-Signal System Design
Alexander de Graaf, EEMCS-CAS
17-5-2011
Outline

1. SystemC-AMS Language Composition
2. Models of Computation
3. Types of Analysis
4. Simulation Control and Tracing
5. Example: Bask Modulator
Acknowledgement

This presentation is derived from:

BDREAMS SystemC-AMS Tutorial, Grenoble 2010
By Karsten Einwich, Fraunhofer IIS/EAS Dresden
1.

SystemC AMS Language Composition
SystemC AMS extension

- The standard package contains:
  - Requirement specification document
  - SystemC AMS extensions User’s Guide

- Can be found on [www.systemc.org](http://www.systemc.org)

- An open source (Apache 2) “proof-of-concept” implementation by Fraunhofer:
  - [http://systemc-ams.eas.iis.fraunhofer.de](http://systemc-ams.eas.iis.fraunhofer.de)
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

- It is **not a replacement of Verilog/ VHDL-AMS or Spice**
- Compared to Matlab, Ptolemy, ... SystemC-AMS supports architectural exploration/refinement and software integration
What’s different between analog and digital?

- Analog equation cannot be solved by the communication and synchronization of processes

- 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)

\[
0 = i_{-t} + \frac{v(w_{-it})}{r_{-it}} + c_{-it} \cdot \frac{d(v(w_{-it}) - v(w_{-p}))}{dt}
\]

\[
0 = \frac{v(w_{-p})}{r_{-prefi}} - c_{-it} \cdot \frac{d(v(w_{-it}) - v(w_{-p}))}{dt}
\]

\[
out = \frac{d}{dt}(k1 \cdot in + k2 \cdot out)
\]
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 Kirchhoff’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 / SC_CTOR**
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
2. 

Models of Computation
1. Timed Data Flow (TDF)
2. Linear Signal Flow (LSF)
3. Electrical Linear Networks (ELN)
2.1

Timed Data Flow (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:

\[
\begin{align*}
\text{s1} &= \text{in} \\
\text{s2} &= f_1(\text{s1}) \\
\text{out} &= f_2(\text{s2})
\end{align*}
\]
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
- delay
- timestep
- Port attribute – number of sample for reading / writing during one module execution
- Port attribute – number of sample delay, number of samples to be inserted while initializing
- 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 simulation phase**

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

- **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

port rates

sca_tdf::sca_signal

ctrl

sca_signal

delay 1

8 kHz

port delay = 1

convertor port

B 1

1

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

cluster timestep

scheduling list creation / calculation cluster timestep

SystemC-AMS

28
Timed Dataflow (TDF) Primitive Module

- Module declaration macros

- Port declarations dataflow ports

- Port declaration converter ports (for TDF primitives only)

- Virtual primitive methods called by the simulation kernel – overloaded by the user defined tdf primitive

- Methods for set/get module activation timestep

- Constructor macro / constructor

```
SCA_TDF_MODULE( <name> )
struct <name>:
    public sca_tdf::sca_module
    
    sca_tdf::sca_in< <T> >,
    sca_tdf_sca_out< <T> >
    sca_tdf::sc_in< <T> >,
    sca_tdf::sc_out< <T> >

    void set_attributes()     
    void initialize()        
    void processing()        
    void ac_processing()     

    void set_timestep(const sca_time&)
    sca_time get_time()

SCA_CTOR(<name>)
    <name>(sc_module_name nm)
```
SystemC-AMS

Structure Timed Dataflow User defined Primitive

```c++
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)
- Set/get number of sample delay
- Set time distance of samples get calculated/propagated time distance
- Get absolute sample time

```c
void set_rate(unsigned long rate)
unsigned long get_rate()

void set_delay(unsigned long nsamples)
unsigned long get_delay()

void set_timestep(const sca_time&)
sca_time get_time_step()

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

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

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

```cpp
const T& read(
    unsigned long sample_id=0)
```

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

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

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

```cpp
... operator= (const T&)
```

```cpp
... 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
    }

    void initialize()
    {
        // initialize delay buffer (first 5 sample read by the 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

1/2

- 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
The equation systems will be represented and calculated by objects:

- \texttt{sca_tdf::sca_ltf\_nd} - Numerator / denominator representation
- \texttt{sca_tdf::sca_ltf\_zp} - Pole-zero representation
- \texttt{sca_tdf::sca_ss} - State space equations

The result is a continuous time signal represented by a “artificial” object (\texttt{sca_tdf::sca\_ct\_proxy} or \texttt{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_1d 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;
        // high or low cut-off freq.
        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

```systemc
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_in<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 imports throughout the hierarchy (each primitive import must be connected to exactly one primitive outport)

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

```plaintext
SC_MODULE

in_tdf

Port-to-port binding

Discrete-event input port

sc_core::sc_in<T>

TDF input port

sca_tdf::sca_in<T>

Port-to-port binding

Discrete-event input port

sc_core::sc_in<T>

TDF input converter port

sca_tdf::sca_de::sca_in<T>

Port-to-port binding

TDF input port

sca_tdf::sca_in<T>

TDF output converter port

sca_tdf::sca_de::sca_out<T>

TDF signal

sca_tdf::sca_signal<T>

Port-to-port binding

Discrete-event output port

sc_core::sc_out<T>

TDF output converter port

sca_tdf::sca_de::sca_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);
  }
};
```
2.2

*Linear Signal Flow (LSF)*
Linear Signalflow (LSF)

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

- LSF models are always hierarchical models

- Ports:
  - \(\text{sca\_lsf}::\text{sca\_in}\) - input port
  - \(\text{sca\_lsf}::\text{sca\_out}\) - output port

- Channel / Signal:
  - \(\text{sca\_lsf}::\text{sca\_signal}\)
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

```c
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);
    }
};
```

Diagram:
- Input: `in`
- Module: `sub1`
- Output: `out`
- Signal: `sig`
- Equation: \[ k \cdot \frac{d}{dt} \dot{y} \]
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
2.3

*Electrical Linear Networks (ELN)*
Electrical Linear Network (ELN)

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

- ELN models are always hierarchical models

- Ports:
  - \textit{sca}\_\textit{eln}::\textit{sca}\_\textit{terminal} - conservative terminal

- Channel / Node:
  - \textit{sca}\_\textit{eln}::\textit{sca}\_\textit{node} – conservative node
  - \textit{sca}\_\textit{eln}::\textit{sca}\_\textit{node}\_\textit{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_vcvs`
- `sca_eln::sca_vccs`
- `sca_eln::sca_ccvs`
- `sca_eln::sca_cccs`
- `sca_eln::sca_nullor`
- `sca_eln::sca_gyrator`
- `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::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`
- `...`
Example: ELN language constructs

```cpp
SC_MODULE(myElnModel) // model using ELN primitive modules {
    sca_el n::sca_terminal in, out; // ELN terminal (input and output)
    sca_el n::sca_node n1; // ELN node
    sca_el n::sca_node_ref gnd; // ELN reference node
    sca_el n::sca_r *r1, *r2;
    sca_el n::sca_c *c1;

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

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

        r2 = new sca_el n::sca_r("r2", 100.0);
        r2->p(n1);
        r2->n(gnd);
    }
```
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 $\sim 5$ times larger than the simulation step width
- Prevent the use of extremely high/low values and large differences in the dimensions
3.

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;

        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);
        }
    }
}

SCA_CTOR(combfilter)
{
    ...
}
```

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

\[
z = e^{j2\pi \frac{f}{f_s}}
\]
4.

Simulation Control and Tracing
Tracing of Analog Signals

- SystemC AMS has its 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_signals`, `sca_eln::sca_node` (voltage)
  - and `sc_core::sc_signals`
  - 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:

%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:
  
  \begin{verbatim}
  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
  \end{verbatim}

- 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
  - \texttt{sc\_start(10.0,SC\_MS);} // run simulation for 10 ms
  - \texttt{sc\_start();} // run simulation forever or \texttt{sc\_stop()} is called

- AC-domain / AC-noise-domain
  - Run simulation from 1Hz to 100kHz, calculate 1000 points logarithmically spaced:
    - \texttt{sca\_ac\_start(1.0,100e3,1000,SCA\_LOG);} \hfill // ac-domain
    - \texttt{sca\_ac\_noise\_start(1.0,100e3,1000,SCA\_LOG);} \hfill // ac-noise domain
  - Run simulation at frequency points given by a \texttt{std::vector<double>}:
    - \texttt{sca\_ac\_start(frequencies);} \hfill // ac-domain
    - \texttt{sca\_ac\_noise\_start(frequencies);} \hfill // ac-noise domain
# include <systemc-ams.h>

int sc_main(int argc, char* argv)
{
    // 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("sctf");
    sc_trace(sctf, scsig1, "scsig1"); ...
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 2ns
satf->disable(); // stop writing
sc_start(2.0, SC_MS); // continue 2ns
satf->enable(); // continue writing
sc_start(2.0, SC_MS); // continue 2ns
// 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
}
Example: BASK De/Modulator
Ack. Markus Damm (TU VIENNA)
What this talk is about

- We walk through a simple communication system example (BASK)
- Along the way
  - we encounter some common pitfalls
  - review some SystemC AMS concepts
- You should get an idea on how
  - to model with SystemC AMS
  - SystemC AMS simulation works
Generating a sine-wave in SystemC-AMS

```cpp
SCA_TDF_MODULE(sine) {
    sca_tdf::sca_out<double> out; // output port
    void processing() { // our workhorse method
        out.write( sin( sc_time_stamp().to_seconds()*(1000.*2.*M_PI))) ;
    }
    SCA_CTOR(sine) {} // constructor does nothing here
};
```

- The `processing()` method specifies the process of the Module.
- In this case, it generates a 1kHz Sine wave.
- However, we used the SystemC method `sc_time_stamp()` to get the current simulation time...
- SystemC AMS has its own method for this, `sca_get_time()`. We will see shortly, what difference this makes...
Instantiating and connecting

```c++
#include "systemc-ams.h"

SCA_TDF_MODULE(drain) { // a drain module to connect the signal to
    sca_tdf::sca_in<double> in; // input port
    SCACTOR(drain) {} // constructor does nothing, no processing() specified!
};

int sc_main(int argc, char* argv[]){
    sc_set_time_resolution(10.0, SC_NS);
    sca_tdf::sca_signal<double> sig_sine ;
    sine sin("sin");
    sin.out(sig_sine);
    sin.out.set_timestep(100,SC_NS); // The sampling time of the port
    drain drn("drn");
    drn.in(sig_sine);
    sca_trace_file* tr = sca_create_vcd_trace_file("tr"); // Usual SystemC tracing
    sca_trace(tr, sig_sine ,"sig_sine");
    sc_start(2, SC_MS);
    return 0;
}
```
...completely as expected, it also worked with \texttt{sc\_time\_stamp()}

So what’s the big deal? Consider the following seemingly innocent change in the \texttt{drain}:

\begin{verbatim}
SCA_TDF_MODULE(drain) {
    sca_tdf::sca_in<double> in;
    void set_attributes(){
        in.set_rate(1000); }
    SCA_CTOR(drain) {} }
\end{verbatim}

The simulation result now looks like this:

No changes were made in the sine module. This is a side effect due to the data rate change in the drain!
Data rates and scheduling

• The explanation is simple: before this change, the process schedule looked like this: sine, drain, sine, drain,...

• Now, the drain reads 1000 token at once, thus, the sine modules’ processing() has to be executed a 1000 times before the drains’ processing() can be executed once. That is, the schedule looks like this: sine, sine,..., sine, drain, sine, sine,..., sine, drain,...

• During those successive executions of the sine modules’ processing(), the sc_time_stamp() method returns the same value every time – yielding the same output every time!

• The sca_get_time() method takes this into account ⇒ Don’t use sc_time_stamp() in TDF-Modules! You might get errors where you don’t have the slightest clue of the cause.
Timed Synchronous Data Flow (TDF)

- The static schedule is simply determined by the data rates set at the ports with `set_rate()`. So far, this is usual TDF.
- In SystemC AMS, a sampling period is associated to token production/consumption of a port with `set_timestep()`.
- ...but it is set only at one port of a cluster!
Simulation time and multirate dataflow

- Although `sca_get_time()` works well globally, there is one more pitfall when using data rates > 1.
- Consider the following simple example:

![Diagram showing simulation time and multirate dataflow]

- Depending on the application, we *might* have to take into account the difference between the value of `sca_get_time()` when a token is read / written and the time the respective token is actually valid.
- This is especially true for token production.
- Let’s see how to apply this knowledge for a bullet-proof sine source with custom data rates...
A sine-wave module with custom data rate

```cpp
SCA_TDF_MODULE(sine) {  
  sca_tdf::sca_out<double> out;
  int datarate; double freq, stepsize; // some data we need
  void set_attributes(){ out.set_rate(rate); }
  void initialize(){      // This method is called when scheduling is done already...
    double sample_time = out.get_timestep().to_seconds(); // ...such that get_T() works.
    stepsize = sample_time*freq*2.*M_PI;
  }
  void processing(){
    for(int i=0; i<rate; i++){
      out.write(sin( sca_get_time().to_seconds()freq*2*M_PI+(stepsize*i) ),i);
    }
  }
}
sine(sc_module_name n, double _freq, int _datarate){  // constructor with
  datarate = _datarate;                                   // additional parameters
  freq     = _freq;
}
};
```

This module is completely self-contained and makes no assumptions on the rest of the model. It will work no matter what.
A BASK modulator demodulator exploiting multirate dataflow

• BASK: Binary Amplitude Shift keying
• Principle of BASK modulation:

![Diagram of BASK modulation](image)

• Principle of BASK de-modulation:

![Diagram of BASK demodulation](image)
The mixer (modulation)

```cpp
SCA_TDF_MODULE(mixer) {
    sca_tdf::sca_in<bool> in_bit;
    sca_tdf::sca_in<double> in_wave;
    sca_tdf::sca_out<double> out;
    int rate;
    void set_attributes(){
        in_wave.set_rate(rate);
        out.set_rate(rate);
    } // NOTE: data rate 1 is the default for in_bit
    void processing(){
        if(in_bit.read()){ // Input is true
            for(int i=0; i<rate; i++) { // => Copy double input to output
                out.write(in_wave.read(i),i);
            }
        }
        else{ // write zeros otherwise
            for(int i=0; i<rate; i++) {out.write(0.,i);} 
        }
    }
    mixer(sc_module_name n, int _rate){rate = _rate;}
};
```
The overall transmitter

```
SC_MODULE(transmitter) {
    sca_tdf::sca_in<bool> in; // The bits modulated onto the carrier
    sca_tdf::sca_out<double> out; // the modulated wave
    mixer* mix;       // a mixer
    sine* sn;         // The source of the carrier wave
    sca_tdf::sca_signal<double> wave;

    transmitter(sc_module_name n, double freq, int rate){
        mix = new mixer("mix", rate);  // Instantiate the mixer with
        mix->in_bit(in);              // the data rate
        mix->in_wave(wave);
        mix->out(out);

        sn = new sine("sn", freq, rate); // Instantiate the carrier source
        sn->out(wave);                  // with frequency and data rate
    }
};
```

**Note:** This is an ordinary hierarchical SystemC module, where the submodules are SystemC AMS modules!
The rectifier

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

    void processing(){
        out.write(abs(in.read()));
    }

    SCA_CTOR(rectifier){}
}
```
The lowpass filter

```c
SCA_TDF_MODULE(lowpass) { // a lowpass filter using an ltf module
    sca_tdf::sca_in<double> in; // input double (wave)
    sca_tdf::sca_out<double> out; // output is the filtered wave
    sca_ltf_nd ltf_1; // The Laplace-Transform module
    double freq_cutoff; // the cutoff-frequency of the lowpass

    sca_util::sca_vector<double> Nom, Denom; // Vectors for the Laplace-Transform module

    void processing(){
        out.write(ltf_1(Nom, Denom, in.read()));
    }

    lowpass(sc_module_name n, double freq_cut){
        Nom(0) = 1.0; Denom(0) = 1.0; // values for the LTF
        Denom(1) = 1.0/(2.0* M_PI* freq_cut); // to describe a lowpass-filter
    }
};
```
Electrical network version of the lowpass filter

```c
SC_MODULE(lp_eln) {  
sca_tdf::sca_in<double> in;  
sca_tdf::sca_out<double> out;  
sca_eln::sca_node n1,n2;  // electrical nodes  
sca_eln::sca_node_ref gnd;  
sca_c c; sca_r r;  // capacitor and resistor  
sca_eln::sca_tdf::sca_vsource vin;  // TDF to voltage converter  
sca_eln::sca_tdf::sca_vsink vout;  // voltage to TDF converter  
lp_eln(sc_module_name n, double freq_cut):c("c"),r("r"),vin("vin"),"vout")  
    double R = 1000.;  // choose fixed R  
    double C = 1/(2*M_PI*R*freq_cut);  // and compute C relative to it  
    vin.p(n1); vin.n(gnd); vin.ctrl(in);  
    vout.p(n2); vout.tdf_voltage(out);  
    c.value = C;  
    c.p(n2); c.n(gnd);  
    r.value = R;  
    r.n(n1); r.p(n2);  
};
```
Bit recovery

**SCA_TDF_MODULE(bit_recov){**

```cpp
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<bool> out;

    int rate, sample_pos;
    double thresh;

    void set_attributes(){
        in.set_rate(rate);
    }

    void processing(){
        if(in.read(sample_pos) > thresh) out.write(true);
        else out.write(false);
    }

    bit_recov(sc_module_name n, double _thresh, int _rate){
        rate = _rate; thresh = _thresh;
        sample_pos = static_cast<int>(2.*(double)rate/3.);  // compute sample position
    }
};
```

- Note that we just read the sample point we are interested in
- All other values are basically discarded!
The overall receiver

```cpp
SC_MODULE(receiver) {  
sca_tdf::sca_in<double> in;
sca_tdf::sca_out<bool> out;
bandpass* bp;
rectifier* rc;
lowpass* lp;
bit_recov* br;
sca_tdf::sca_signal<double> wave1, wave2;
receiver(sc_module_name n, double freq, int rate, double thresh) {
  rc = new rectifier("rc");
  rc->in(in);
  rc->out(wave1);
  lp = new lowpass("lp", freq/3.);
  lp->in(wave1);
  lp->out(wave2);
  br = new bit_recov("br", thresh, rate);
  br->in(wave2);
  br->out(out);
}
};
```
#include "systemc-ams.h"

int sc_main(int argc, char* argv[]){
    sc_set_time_resolution(10.0, SC_NS);

    sca_tdf::sca_signal<bool> bits, rec_bits; // the bits which are transmitted & received
    sca_tdf::sca_signal<double> wave; // the modulated wave

    bitsource bs("bs"); // The data source
    bs.out(bits);
    bs.out.set_timestep(1, SC_MS);

    transmitter transmit("transmit", 10000., 1000);
    transmit.in(bits);
    transmit.out(wave);

    receiver receiv("receiv", 10000., 1000, 0.02);
    receiv.in(wave);
    receiv.out(rec_bits);

    drain drn("drn");
    drn.in(rec_bits);

    sca_trace_file* tr = sca_create_vcd_trace_file("tr");
    ...
    sc_start(20, SC_MS);
    return 0;}
• Looks fine! However, something is strange... who knows what it is?
• Multirate-dataflow allowed us to overcome causality!
  • The bit recovery module reads the sample of interest during the same **processing**() execution when it also writes the result.
  • However, the output token is valid the same time as the first input token.
Using delay to regain causality

```systemc
SCA_TDF_MODULE(bit_recov) {
    ...
    void set_attributes(){
        in.set_rate(rate);
        out.set_delay(1);
        ...
    }
};
```

- This delays the output of the bit recovery module by one token, which in this case results in a 1 ms delay.
- Delays also have to be used in the presence of feedback loops.
- You can also write initial values in the `initialize()` method.
A simple environment model

```c++
SCA_TDF_MODULE(environment) {
  sca_tdf::sca_in<double> in1, in2;
  sca_tdf::sca_out<double> out;
  double attenuation, variance;

  void processing() {
    out.write((in1.read()+in2.read())*attenuation+gauss_rand(variance));
  }

  environment(sc_module_name n, double _attenuation, double _variance) {
    variance    = _variance;
    attenuation = _attenuation;
  }
};
```

- This module takes two waves, adds them and exposes them to attenuation and Gaussian noise.
- We assume the presence of a Gaussian noise function here.
Simulation result with environment model
Simulation result with environment model
Simulation Environment

• We mostly use a very simple simulation environment, which is completely open source & free:
  • Linux (Suse, Ubuntu), Cygwin
  • VIM with custom Syntax highlighting (but any editor will do)
  • Makefiles
  • GTKWave (waveform viewer)
• In SystemC teaching, we encourage the students to install this environment on their own desktop computer / laptop
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${SYSTEMC_PATH}/include \n   -I${SYSTEMC_AMS_PATH}/include

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

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

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

uppercase i

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

lowercase L

!!! Library order Important !!!
```
More ...

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