May 2006

SystemC-AMS 0.15

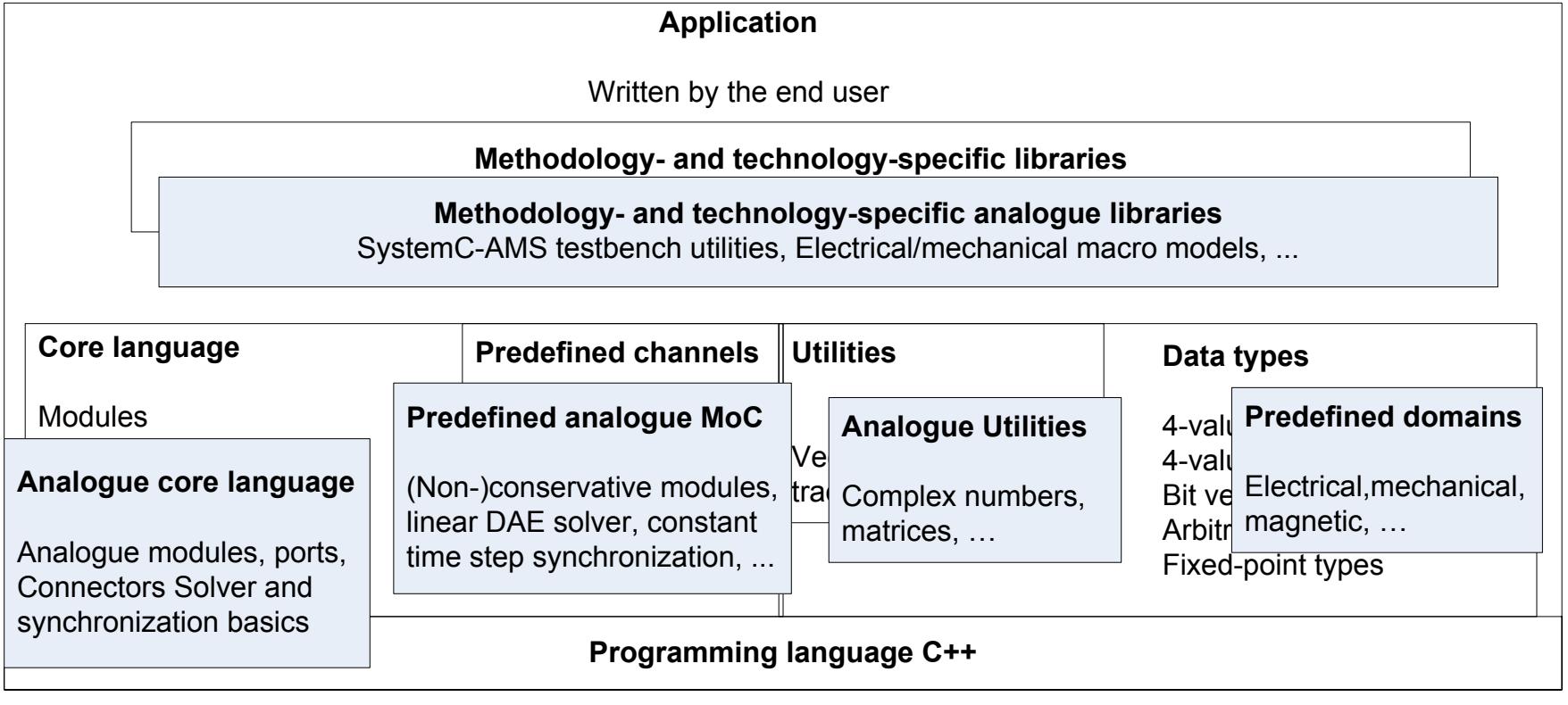
Fraunhofer Institut
Integrierte Schaltungen

Focus of SystemC-AMS

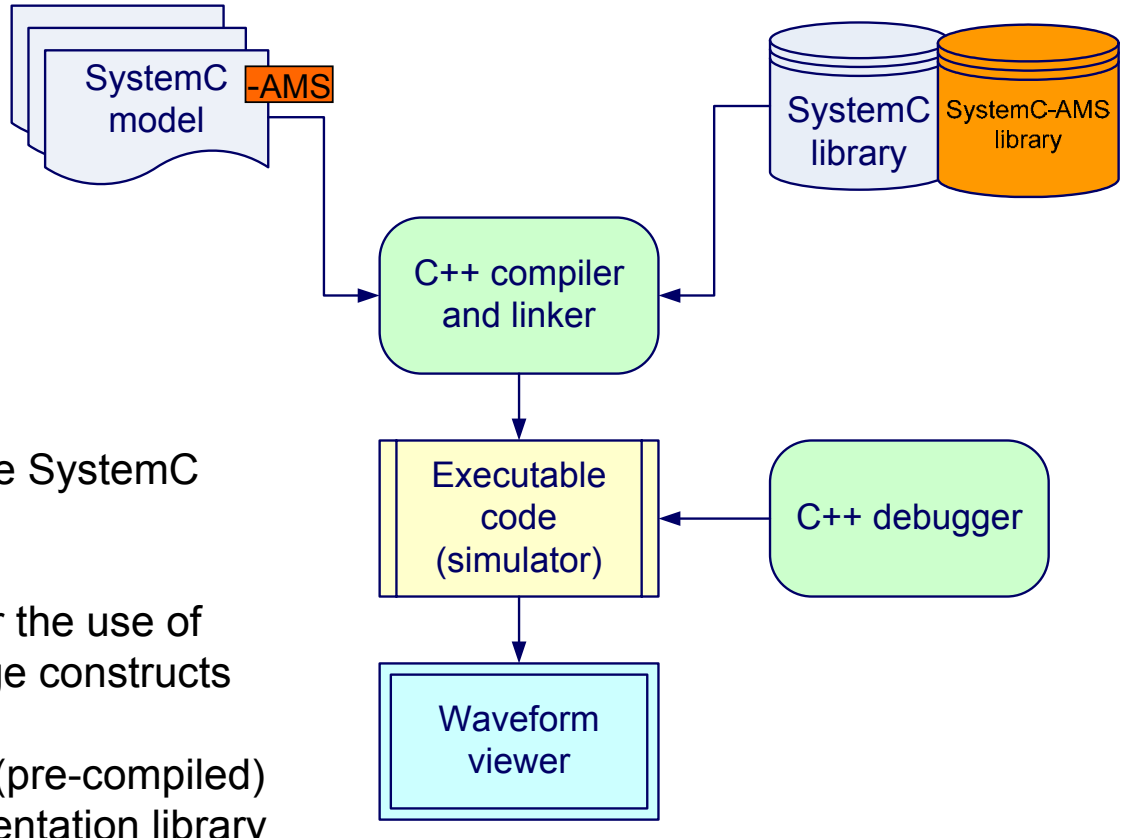
Description, Simulation and Verification for:

- Functional **Complex** integrated systems
- **A**nalog **M**ixed **S**ignal systems / **Heterogeneous** systems
- **Specification / Concept engineering**
- **System design,** Reference (“golden”) model development
- Embedded **software** development
- Next Layer (driver) software development
- **Customer** model

SystemC – AMS on top of SystemC

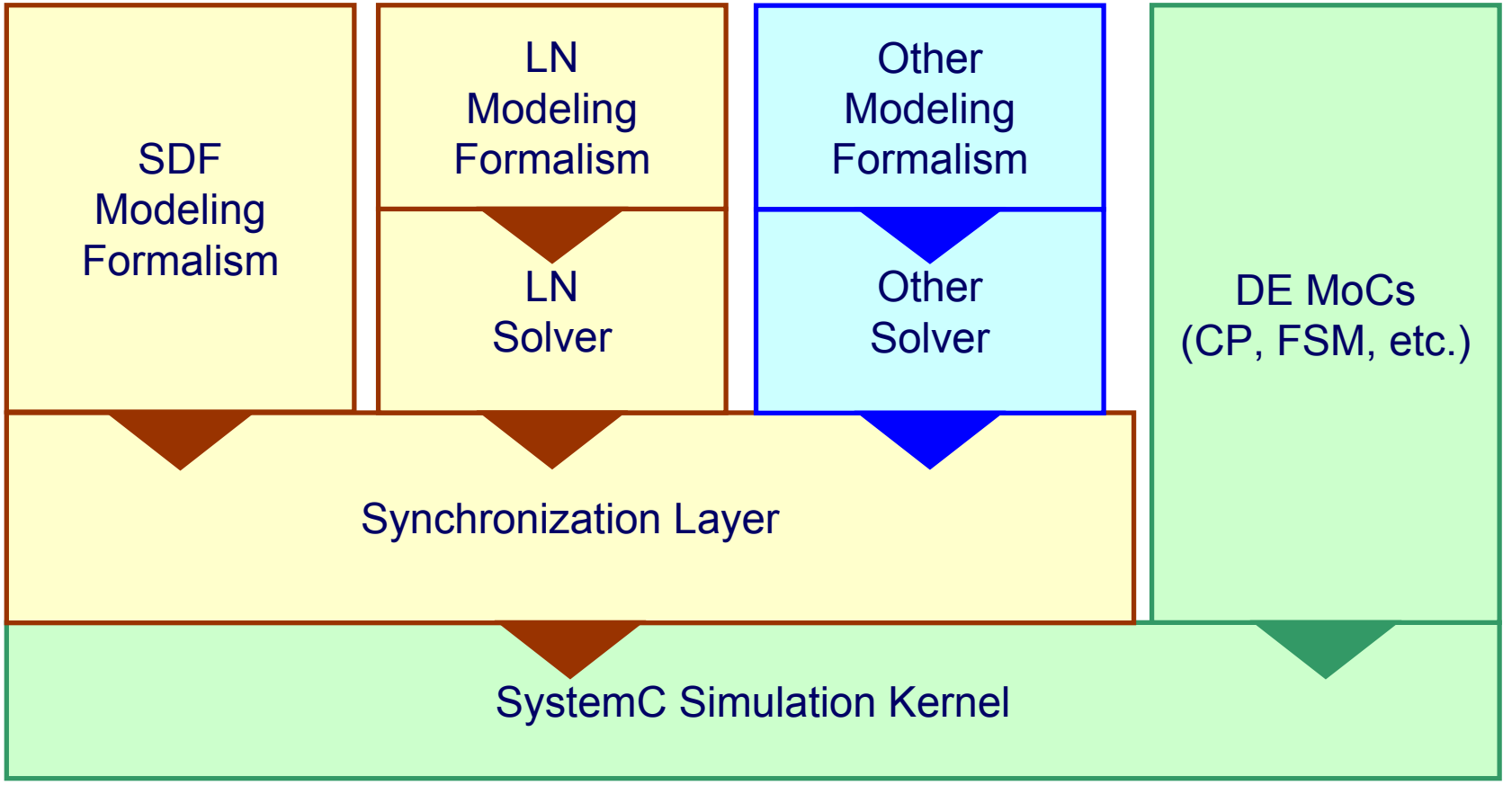


SystemC-AMS is an extension of SystemC

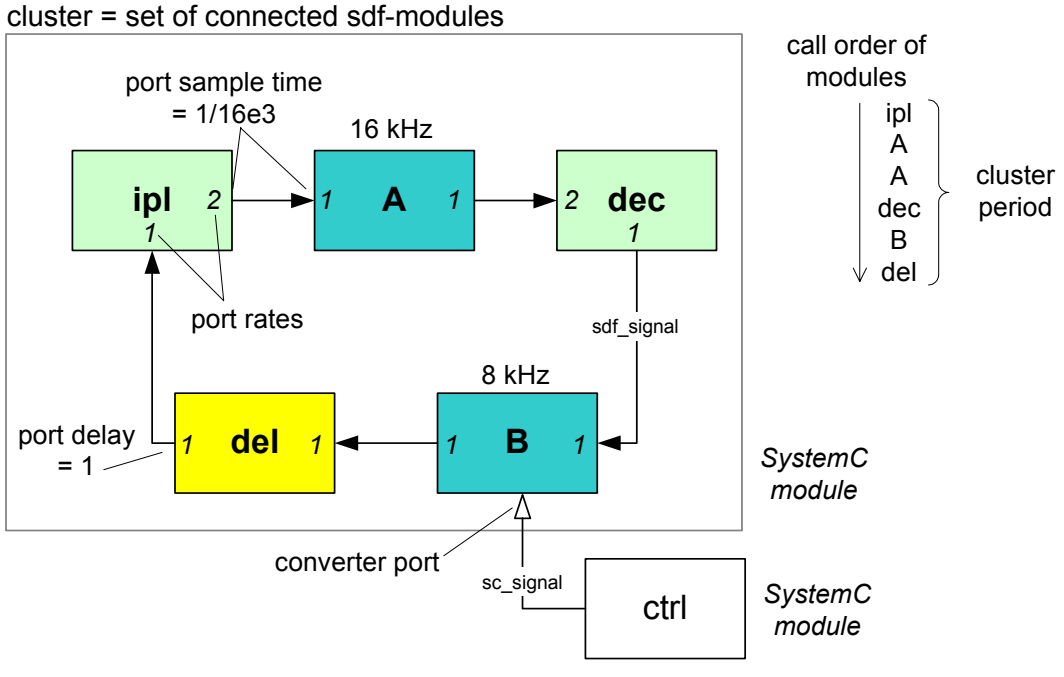


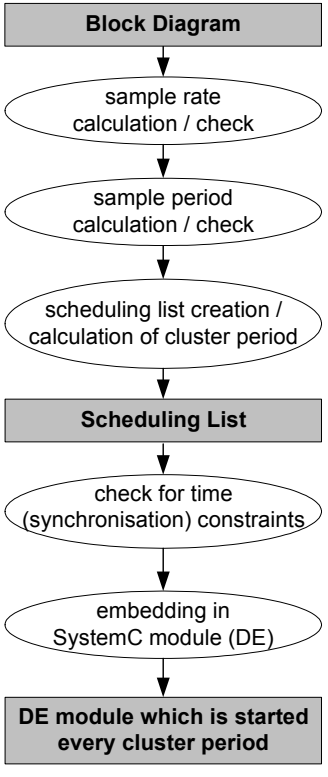
- No changes of the SystemC language
- No restrictions for the use of SystemC language constructs
- Use of the same (pre-compiled) SystemC implementation library

SystemC-AMS architecture



Synchronous Multi-Rate Data Flow Implementation (SDF) for Modelling Non-Conservative Analogue Systems and digital Signal Processing





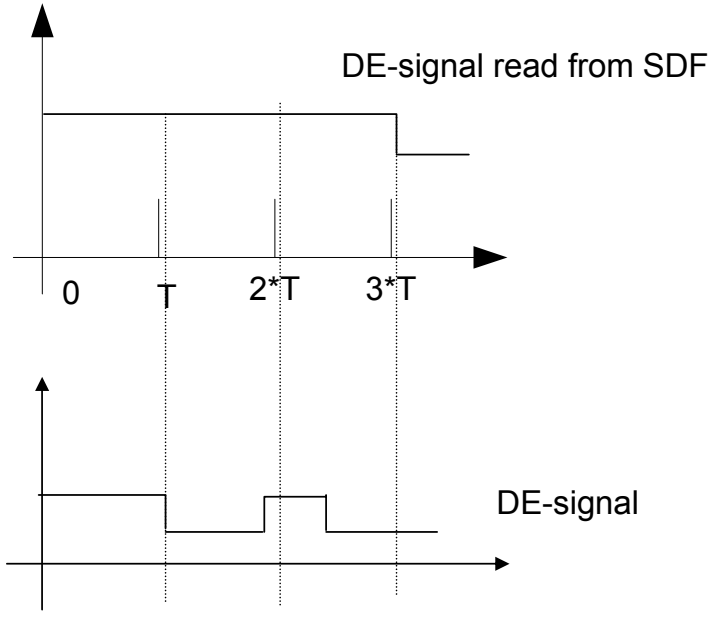
Special Properties of SystemC-AMS SDF-Domain

- SDF-sample are mapped to sc_time

- DE-Signals will be sampled at delta 0 of the specified time point and scheduled also at delta 0 (and thus valid at least at delta 1)

- Sample time is specified as port – attribute and propagated through the cluster

- Thus the sample time distance must be at least specified at one port of a module in a SDF cluster – if more attributes given the simulator performs a consistency check



Syntax Example of SDF Module

module declaration

port declarations

set port attributes

Initialisation

process method

post processing

constructor

```
SCA_SDF_MODULE(delay_sdf)
{

  sca_sdf_in<double>  in;
  sca_sdf_out<double> out;


  void attributes() {
                        out.set_delay(1);
                      }

  void init() {
                 out.write(0.0); // initial value
               }


  void sig_proc()  {
                    out.write(in.read());
                   }

  void post_proc()  {
                       cout << name << " done" << endl;
                     }


  SCA_CTOR(delay_sdf){ }
};
```

Language Elements Basic SDF Modules

- Module declaration macros
- Port declarations dataflow ports
- Converter ports to connect SystemC-signals (the SystemC signals will be sampled)
- Virtual primitive dataflow methods, called automatically by schedule
- Constructors (currently the same like SC_CTOR – however use SCA_CTOR for compatibility with future extensions)

SCA_SDF_MODULE(*<name>*)

sca_sdf_out < *<type>* > // outport
sca_sdf_in < *<type>* > // inport

sca_scsdf_in < *<type>* >
sca_scsdf_out < *<type>* >

void attributes() // before elaboration
void init() // before simulation start
void sig_proc() // during simulation (mandatory)
void post_proc() // after simulation

sc_time sca_get_time() //gives the absolute time of the current module call

SCA_CTOR(*<name>*) |
<name>(sc_module_name nm)

Virtual Methods of a Basic SDF Module

- void attributes();
  - Used to set attributes which influence the elaboration – especially port attributes like rate, delay, T
- void init()
  - Initialize module members, delays, … - is called immediately before simulation start, after the elaboration is done, the analyzed port attributes (T) are accessible
- void sig_proc()
  - Implements module behavior, the only mandatory method
- void post_proc()
  - Is called immediately before the executable exits (currently if the sca_terminate() function is called), the data structures still exist, can be used for post processing issues like FFT

Attributes of SDF - Ports

- Rate is the number of samples which has to be read / written per module (sig_proc) invocation, the default value is 1, has to be set in attributes
- Delay sets the number of samples which has to be written during initialization, this is the number of sample delay or the number of sample periods delay, the default value is 0 , has to be set in attributes
- T is the time distance between sample, T has to be set at least at one port per dataflow cluster, default is unset , has to be set in attributes
- get_T() returns the analyzed sampling time, may be called earliest in init()

```
port.set_rate(2);


outport.set_delay(1);

port.set_T(double, sc_time_unit);
port.set_T(sc_time);


port.get_T();  // get analysed sample period as sc_time
port.get_T().to_seconds();  // in seconds

port.get_t0();    // get analysed first time point as sc_time

port.get_rate();  //get set rate as long
port.get_delay(); //get set delay as long
port.get_time(); //get absolute time of the first sample of
                 //the current call as sc_time
port.get_sample_cnt(); //absolute number of sample without
                       //sample of the current call
```

SDF - Port Access

- Inport (sca_sdf_in<type> or sca_scsdf_in<type>) read for the first sample, can be called in sig_proc() and post_proc() and only for inports, consecutive calls at the same module evaluation will return the same value
- Outport (sca_sdf_out<type> or sca_scsdf_out<type>) write to the first sample of the current invocation, if the delay >= 1 a sample can be written in init() otherwise only in sig_proc(), consecutive writes in the same invocation overwrite the sample (last write of invocation wins)
- Read of the n-th. sample of the current invocation from an inport, is only allowed in sig_proc() and post_proc(), n must be smaller than the set rate consecutive calls at the same invocation will return the same value
- Writes the n-th. sample of the current block call to an outport, if the delay > n the n-th. delay initialization sample can be written during init(), otherwise the access is only allowed in sig_proc() if n smaller the set rate, consecutive calls in the same invocation to the same sample overwrite the sample (last write of an invocation wins)
- Growing n means growing time (fifo)

```
tmp=port; //tmp is variable of type <type>
tmp=port.read();              // recommended


port=tmp;
port.write(tmp);              // recommended


tmp=port[n];                  //n is a long number
tmp=port.read(n);             // recommended


port[n]=tmp;
port.write(tmp, n);           // recommended
```

Hierarchical SDF Modules

- Signals/SDF-channel to connect dataflow ports
- Module declaration – standard SystemC macro – no difference
- Constructor for this declaration – standard SystemC constructor – no difference, SC_HAS_PROCESS is not required
- Module instantiation like SystemC – no difference
- Port to Port binding same like SystemC – converter ports of a primitve have to be bound to the converted port kind (e.g. a sca_scsdf_in<int> to a sc_in<int> port) see converter ports for details
- Port to signal binding same like SystemC - converter ports of a primitve have to be bound to the converted signal kind (e.g. a sca_scsdf_in<int> to a sc_signal<int> port) see converter ports for details

sca_sdf_signal< <*type*> > signal1;

SC_MODULE(<*name*>)

SC_CTOR(<name>) |
<name>(sc_module_name nm)

module1 my_inst1(“my_inst1“);
module1* my_inst2=new module1(„my_inst1“);

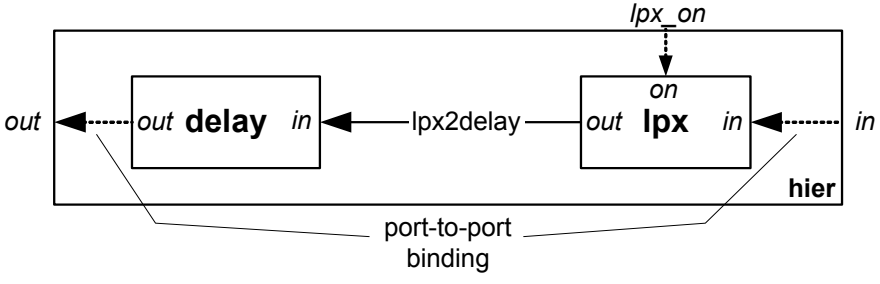
my_inst1.port(parent_port);
my_inst2->port(parent_port);

my_inst1.port(signal1);
my_inst2->port(signal1);

Hierarchical SDF Modules - Rules

- ... are just logical sets of basic modules. The scheduler ignores any hierarchy and runs on the „flat“ system
- An hierarchical port is a port of a hierarchical module (here considered from inside the module) – the port is described with the same port classes like basic module ports – except converter ports
- Ports of hierarchical modules can not have attributes like delays and sample rates
- Hierarchical modules can’t have converter ports – they use sc_in / sc_out ports instead
- A basic module’s outport must be connected at least with one primitive inport (may crossing some hierarchical ports – which will be ignored)
- A basic module’s inport must be connected with exactly one outport (also may crossing hierarchical ports)
- Every Outport/Inport must be connected with exactly one sdf-channel (a sca_sdf_signal)
- A hierarchical inport must bound to one or more inports
- An hierarchical outport must bound to exactly one outport and may be to any number of inports
- A hierarchical port can’t be connected to a signal (e.g sca_sdf_signal) inside the module

Hierarchical Module Example




```
SC_MODULE(hier)
{
  sca_sdf_in<double>    in;
  sca_sdf_out<double> out;

  sc_in<bool>              lpx_on;

  delay*  delay_i;
  lpx*    lpx_i;

  sca_sdf_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;
  }
};
```

“Analog” Representation of linear dynamic Systems

- Transfer function

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

  - Easy extraction from networks, ...

- Zero-Pole representation

$$H(s) = k \bullet \frac{(s - z_0) \bullet (s - z_1) \bullet \ldots \bullet (s - z_n)}{(s - p_0) \bullet (s - p_1) \bullet \ldots \bullet (s - p_n)}$$

  - Operational amplifier, analog filters

- State Space equations

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

  - Good state control

Analog Linear Behavioral Models

- Transfer function

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

- Zero Pole

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

- State Space

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

out=**sca_ltf_1**(NUM, DEN, S, input);

out=**sca_zp_1**(Z, P, K, S, input);

out=**sca_ss_1**(A, B, C, D, S, input);

Zp and ss not yet implemented

Analog Linear Behavioral Models

- The class sca_ltf_nd | sca_ltf_zp | sca_ss can be used in any (and only) sca_module context (as a sca_module member)
- Currently only sca_sdf_module's available for users – thus the classes can be currently used only in synchronous dataflow models
- The class instantiates a corresponding equation system
- The sca_vector<double> S stores the state of the corresponding equation system
- The sca_vector<double> (e.g. A, B for ltf) or sca_matrix<double> (e.g. A, C for ss) holding the parameters of the system representation
- If the parameter were accessed the equation system will be re-initialized – if you want change the parameters do not access them during simulation
- The state vector is not changed during re-initialization – however his representation is not defined (except state space)
- For the classes the () operator with the required parameters is defined
- The used time step is always the calling period of the parent sca_module – the equation system state is stored in S – thus it is possible to create instances with different parameters and realize e.g. a cut off frequency switch simply by using the instances alternatively with the same S – thus no re-initialization is required

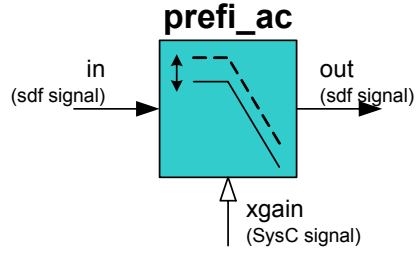
```
sca_ltf_nd ltf_1;  //declaration of ltf-instance
//bracket operator for ltf return value of type double
(double) ltf_1 (sca_vector<double>& NUM,
                sca_vector<double>& DEN,
                sca_vector<double>& STATE, //optional – default
     reset state after coefficient change
                 double                 in  ,
                  unsigned long n=1); //internal oversampling rate
//alternatively a method calc() with same arguments can be used

sca_ltf_zp zp_1;  //declaration of ltf-instance
//breaket operator for ltf return value of type double
(double) zp_1 (sca_vector<double>& Z,
                sca_vector<double>&  P,
                double                  k,
                sca_vector<double>&  S,
                double                 in ,
                 unsigned long n=1 ); //internal oversampling rate
//alternatively a method calc() with same arguments can be used

sca_ss ss_1;  //declaration of ltf-instance
//breaket operator for ltf return value of type sca_vector<double>
(sca_vector<double> ) ss_1
              ( sca_matrix<double>& A, sca_matrix<double>&  B,
                sca_matrix<double>& C, sca_matrix<double>&  D,
                sca_vector<double>& S
                sca_vector<double>&  in ,
                unsigned long n=1 ); //internal oversampling rate
//alternatively a method calc() with same arguments can be used
```

Analog Linear Behavioral Models Example

```
SCA_SDF_MODULE(prefi_ac)
{
  sca_sdf_in<double>  in;
  sca_sdf_out<double> out;
  sca_scsdf_in<bool> xgain;

  // parameter
  double prefi_fc;     //cut-off frequency
  double prefi_gain0;  //gain if !xgain
  double prefi_gain1;  //gain if  xgain

  // states
  sca_ltf_nd  ltf_1; //filter equation instance
  sca_vector<double> A, B;   //coeff vector
  sca_vector<double> S;      //state vector

  void init() {
    //filter coeffs for transfer function
    B(0) = 1.0;
    A(0) = 1.0;
    A(1) = 1.0/(2.0*M_PI*prefi_fc);
  }
```

```
void sig_proc() {
 double tmp = ltf_1(B,A,S,in.read());

 if (xgain.read())  out.write(tmp * prefi_gain1);
 else               out.write(tmp * prefi_gain0);
            }

  SCA_CTOR(prefi_ac) {
    // defaults
    prefi_fc     = 1.0e6;
    prefi_gain0  = 2.74;
    prefi_gain1  = 2.74 * 2.2;
  }
};
```

$$H(s) = \frac{1}{1 + \frac{1}{2\pi f_c} s}$$



Small Signal Frequency Domain Specification - Overview

- Every dataflow module can have a small signal frequency domain specification/implementation
- This specification describes a complex transfer function from the dataflow inports to the dataflow outports
- This description is independent from the port type – the only condition is that the in-/outports are dataflow ports (or in general a sca_port on which a sca_channel is connected which has a ac_domain semantic)
- There is no automatic check, that the frequency domain specification is consistent with the time domain implementation
- For dataflow outports without ac-domain implementation assumed as zero – dataflow models without ac-domain implementation will not lead to an error
- Time and frequency domain simulation can be mixed
- The current time domain values can be read using the time domain access methods (especially for control ports)
- For linear networks the frequency domain specification is defined automatically in the primitives
- During ac-analysis the resulting linear complex equation system is solved for the given frequency points
- The signals / wires added for tracing are valid for ac-domain also
- If you use time domain and ac-domain simulation consecutively you have to redirect the file stream to different files or enable/disable the trace-stream (see tracing)

Frequency Domain Specification

- Virtual method for module complex transfer function
- Dataflow Port access within ac_domain
- Frequency access within ac_domain
- Linear electrical networks have an implicit frequency domain representation
- Global function to start ac domain simulation from testbench

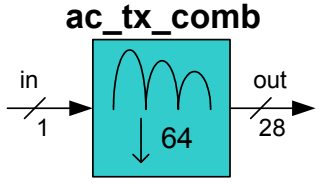
ltf_zp and ss not yet implemented

```
void ac_sig_proc()

bool sca_ac_domain(); //true if ac-domain (noise also) running

sca_complex tmp=sca_ac(<inport name>);
sca_ac(<outport name>) = tmp;


double tmp=sca_ac_freq();  //current frequency in Hz
double tmp=sca_ac_w();     //current frequency in 1/s

sca_complex tmp=sca_ac_delay(double dt);
sca_complex tmp=sca_ac_z(double T, long n=1);
sca_complex tmp=sca_ac_s(long n=1);
sca_complex tmp=sca_ac_ltf_nd(  sca_vector<double> NUM,
                                sca_vector<double> DEN,
                                sca_complex in);

sca_complex tmp=sca_ac_ltf_zp(   sca_vector<double> Z,
                                 sca_vector<double> P,
                                 double             K,
                                 sca_complex in);

sca_vector<sca_complex> tmp=sca_ac_ss(    sca_matrix<double> A,
                                          sca_matrix<double> B,
                                          sca_matrix<double> C,
                                          sca_matrix<double> D,
                                           sca_vector<sca_complex> in);

sca_ac_domain_simulate( double startf,
                        double endf,
                          unsigned long npoints,
                         enum  {SCA_LIN | SCA_LOG});

sca_ac_domain_simulate( vector<double> frequencies);
```

Frequency Domain Specification - Example

```
SCA_SDF_MODULE(ac_tx_comb)
{
 sca_sdf_in<bool>            in;
 sca_sdf_out<sc_int<28> > out;

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


 void ac_sig_proc()
 {
  sca_complex z;
  z = sca_ac_z(in.get_T().in_seconds() , 1 );
  double          k = 64.0; //decimation factor
  double          n = 3.0;  //order of comb filter

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

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

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

  SCA_CTOR(ac_tx_comb) {
    ...
  }
};
```

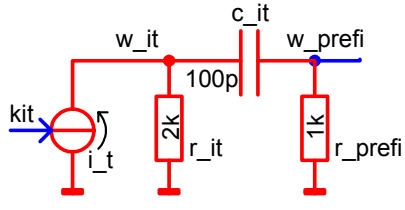


$$H(z) = \left(\frac{1 - z^{-k}}{1 - z^{-1}} \right)^{n} \qquad z = e^{j2\pi f / f_s}$$

Linear Electrical Networks - Example

```
sca_elec_node        w_it;       //electrical node
sca_elec_node        w_prefi;  //converter signal
sca_elec_ref         gnd;       //reference node
sca_sdf_signal<double> kit, s_prefi;

sca_sdf2i i_t(„i_t“);
  i_t.p(gnd);      // pos
  i_t.n(w1);       // neg
  i_t.sdf_ctl(kit);      // current value by signal

sca_r r_it(„r_it“);
  r_it.value = 2e3;
  r_it.p(w_it);
  r_it.n(gnd);

sca_r r_prefi(„r_prefi“);
  r_prefi.value=1e3;
  r_prefi.p(w_prefi);
  r_prefi.n(gnd);

sca_c c_it(„c_it“);
  c_it.value=100e-9;
  c_it.p(w_it);
  c_it.n(w_prefi);

sca_v2sdf conv1(„conv1“);
  conv1.p(w_prefi);
  conv1.sdf_voltage(s_prefi);
```

```
// signal tracing

sca_trace_file* tr1 =
                    sca_create_tabular_trace_file("tr1.dat");


sca_trace(tr1,w_it,“w_it“); //node voltage
sca_trace(tr1,r_it,“w_it“); //current through r_it
```



Linear Electrical Networks

- For connected network instances one equation system will be set up
- For this clustering connections via the reference node (gnd) will be ignored
- Every network must be connected at least with one reference node
- Networks will be re-initialized if a matrix_stamp has changed (e.g. a changeable resistor used for switches is implemented)
- All connection to the SDF domain must be on the same rate
- The network elements are predefined

Linear electrical networks – Network Description

- Module Declaration
- Constructor
- Electrical node
- Electrical reference node
- Electrical port

```
SC_MODULE(<name>)

SC_CTOR(<name>)
<name>(sc_module_name nm)


sca_elec_node          w1;



sca_elec_ref           gnd;


sca_elec_port          p1;
```

Linear Electrical Network Elements(1)

- Resistor

- SDF Controlled Resistor / SC Controlled Resistor
  - A value change re-initializes the equation system

```
sca_r      r1("r1", value=0.0);
sca_r      r1("r1");
  r1.p(w1);
  r1.n(w2);
  //overwrites constructor value
  r1.value = <value>;

sca_sdf2r r2("r2");
  r2.p(w1);
  r2.n(w2);
  r2.ctrl(sdf_sig); // sca_sdf_signal<double> sdf_sig;

sca_sc2r r2("r2");
  r2.p(w1);
  r2.n(w2);
  r2.ctrl(sc_sig); // sc_signal<double> sc_sig;
```

Linear Electrical Network Elements(2)

- Capacitor

- Inductivitor

```
sca_c          c1("c1", <value>);
sca_c          c1("c1");
  c1.p(w1);
  c1.n(w2);
  //overwrites constructor value
  c1.value = <value>;

sca_sdf2c c2("c2");
  c2.p(w1);
  c2.n(w2);
  c2.ctrl(sdf_sig); // sca_sdf_signal<double> sdf_sig;

sca_sc2r r2("c2");
  c2.p(w1);
  c2.n(w2);
  c2.ctrl(sc_sig); // sc_signal<double> sc_sig;

sca_l          l1("l1", <value>);
sca_l          l1("l1");
  l1.p(w1);
  l1.n(w2);
  //overwrites constructor value
  l1.value = <value>;
sca_sdf2l l2("l2");
  l2.p(w1);
  l2.n(w2);
  l2.ctrl(sdf_sig); // sca_sdf_signal<double> sdf_sig;

sca_sc2l l2("l2");
  l2.p(w1);
  l2.n(w2);
  l2.ctrl(sc_sig); // sc_signal<double> sc_sig;
```

Linear Electrical Network Elements(3)

- SDF Controlled Current Source

- SDF Controlled Voltage Source

```
sca_sdf2i  i1("i1", <gain>);
sca_sdf2i  i1("i1"); //gain default 1.0
  i1.p(w1);
  i1.n(w2);
  i1.ctrl(sdf_signal);
  //overwrites constructor value
  i1.gain = <value>;

sca_sdf2v  v1("v1", <gain>);
sca_sdf2v  v1("v1"); //gain default 1.0
  v1.p(w1);
  v1.n(w2);
  v1.ctrl(sdf_signal);
  //overwrites constructor value
  v1.gain = <value>;
```

Linear Electrical Network Elements(4)

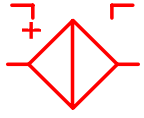
- SC Controlled Current Source

- SC Controlled Voltage Source

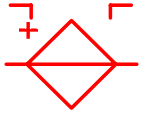
```
sca_sc2i  i1("i1", <gain>);
sca_sc2i  i1("i1"); //gain default 1.0
   i1.p(w1);
   i1.n(w2);
   i1.ctrl(sc_signal); //sc_signal<double>
   //overwrites constructor value
   i1.gain = <value>;

sca_sc2v  v1("v1", <gain>);
sca_sc2v  v1("v1"); //gain default 1.0
   v1.p(w1);
   v1.n(w2);
   v1.ctrl(sc_signal); //sc_signal<double>
   //overwrites constructor value
   v1.gain = <value>;
```

Linear Electrical Network Elements(5)

- Voltage Controlled Current Source



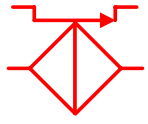
- Voltage Controlled Voltage Source



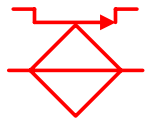
```
sca_vccs vccs1(„vccs1“, K);
sca_vccs vccs1(„vccs1“);
    vccs1.np(w1);
    vccs1.nn(w2);
    vccs1.ncp(w3);
    vccs1.ncn(w4);
    vccs1.value = 1.0;



sca_vcvs vcvs1(„vcvs1“, K);
sca_vcvs vcvs1(„vcvs1“);
    vcvs1.np(w1);          // pos
    vcvs1.nn(w2);          // neg
    vcvs1.ncp(w3);         // pos ctrl
    vcvs1.ncn(w4);         // neg ctrl
    vcvs1.value = 1.0;
```

Linear Electrical Network Elements(6)

- Current Controlled Current Source



- Current Controlled Voltage Source



```
sca_cccs cccs1(„cccs1“, K);
sca_cccs cccs1(„cccs1“);
    cccs1.np(w1);
    cccs1.nn(w2);
    cccs1.ncp(w3);
    cccs1.ncn(w4);
    cccs1.value = 1.0;



sca_ccvs ccvs1(„ccvs1“, K);
sca_ccvs ccvs1(„ccvs1“);
    ccvs1.np(w1);          // pos
    ccvs1.nn(w2);          // neg
    ccvs1.ncp(w3);         // pos ctrl
    ccvs1.ncn(w4);         // neg ctrl
    ccvs1.value = 1.0;
```

Linear Electrical Network Elements(7)

- Constant current source

```
sca_iconst vconst1(„iconst1“, value);
sca_iconst vconst1(„iconst1“);
  iconst1.p(w1);
  iconst1.n(w2);
  iconst1.value = 1.0;
```

- Sinwave current source

```
sca_isin isin1(„isin1“);
   isin1.p(w1);              // pos
   isin1.n(w2);              // neg
   isin1.freq=1e3;           // frequency in Hz
   isin1.ampl=1.0;           // magnitude in V
   isin1.off = 0.0;          //offset
   isin1.phase=0.0;          //phase in degree
   isin1.ac_ampl   = 0.0;
   isin1.ac_phase = 0.0;
   isin1.delay = 0.0;  //delay in seconds
   isin1.delta = 0.0; //damping factor 1/sec
```

Linear Electrical Network Elements(8)

- Constant voltage source

```
sca_vconst vconst1(„vconst1“, value);
sca_vconst vconst1(„vconst1“);
  vconst1.p(w1);
  vconst1.n(w2);
  vconst1.value = 1.0;
```

- Sinwave voltage source

```
sca_vsin vsin1(„vsin1“);
   vsin1.p(w1);            // pos
   vsin1.n(w2);            // neg
   vsin1.freq=1e3;         // frequency in Hz
   vsin1.ampl=1.0;         // magnitude in V
   vsin1.off = 0.0;        //offset
   vsin1.phase=0.0;        //phase in degree
   vsin1.ac_ampl   = 0.0;
   vsin1.ac_phase = 0.0;
   vsin1.delay = 0.0;  //delay in seconds
    vsin1.delta = 0.0; //damping factor 1/sec
```

Linear Electrical Network Elements(9)

- Switch as changeable resistor

```
sca_sc_rswitch switch1(„switch1“);
  switch1.p(w1);
  switch1.n(w2);
  switch1.ctrl(sc_sig); //sc_signal<bool> sc_sig;
  switch1.off_val = false;  //ctrl=true -> on
  switch1.ron   = 1.0e-6; //on-resistance
  switch1.roff   = 1.0e12; //off-resistance
```

- Converter Current to SDF (p – n short cut)

```
sca_sdf_rswitch switch1(„switch1“);
  switch1.p(w1);
  switch1.n(w2);
  switch1.ctrl(sdf_sig); //sca_sdf_signal<bool> sc_sig;
  switch1.off_val = false;  //ctrl=true -> on
  switch1.ron   = 1.0e-6; //on-resistance
  switch1.roff   = 1.0e12; //off-resistance
```

```
sca_i2sdf iconv1(„iconv1“);
   iconv1.p(w1);              // pos
   iconv1.n(w2);              // neg
   iconf1.sdf_current (sdf_sig); // sca_sdf_signal<double>
   iconv1.scale=1.0;          // scaling factor
```

Linear Electrical Network Elements(10)

- Converter voltage between p and the reference to SDF

```
sca_v2sdf iconv1(„vconv1“);
   vconv1.p(w1);          // pos
   vconv1.sdf_voltage(sdf_sig); // sca_sdf_signal<double>
   vconv1.scale=1.0;    // scaling factor
```

- Converter voltage between p and n to SDF

```
sca_vd2sdf vdconv1(„vdconv1“);
   vdconv1.p(w1);         // pos
   vdconv1.n(w2);         // neg
   vdconf1.sdf_voltage (sdf_sig); // sca_sdf_signal<double>
   vdconv1.scale=1.0;  // scaling factor
```

Linear Electrical Network Elements(11)

- Nullor

```
sca_nullor null1(„ null1“);
   null1.nip(w1);          // positive terminal nullator
   null1.nin(w2);          // negative terminal nullator
   null1.nop(w3);          // positive terminal norator
   null1.non(w4);          // negative terminal norator
```

- Gyrator

```
sca_gyrator gyr1(„ gyr1“);
 gyr1.p1(w1);              // positive terminal primary port
 gyr1.n1(w2);              // negative terminal primary port
 gyr1.p2(w3);              // positive terminal secondary port
 gyr1.n2(w4);              // negative terminal secondary port
 gyr1.g1 = 1.0;            //primary gyration conductance in S
 gyr2.g2 = 1.0;            //secondary gyration conductance in S
```

- Ideal Transformer

```
sca_ideal_transformer traf1(“traf1”);
  traf1.p1(w1);
  traf1.n1(w2);
  traf1.p2(w3);
  traf1.n2(w4);
  traf1.n = 1;      //turns ratio
```

SystemC-AMS Signal Tracing

- Open a trace stream
- Switch tracing temporarily off/on to serve disk space and increase performance
- Set different modes for tracing e.g. reduce number of traced samples and format of ac-simulation results
- Close file and open a new file e.g. for storing AC-simulation results in a different file
- Add a signal / wire or instance to a trace

```
sca_trace_file* tr;
      tr=sca_create_tabular_trace_file(string name); |
      tr=sca_create_tabular_trace_file(ostream* out_stream); |

tr1->enable();   //enable tracing (default enabled)
tr1->disable();  //disable tracing can be called at arbitrary time

tr1->set_mode(sca_trace_mode::decimate(<n>)); //writes only evry n-th. time point
tr1->set_mode(sca_trace_mode::sample(period,start_time)); //writes equidistant
    timepoints given by optional start_time (type sc_time, feault SC_ZERO_TIME)
    and period (type sc_time)
tr1->set_mode(sca_trace_mode::ac_abs_phase); //writes ac-simulation result as
    absolute value in dB and phase in degree instead of complex numbers (default)
tr1->set_mode(sca_trace_mode::ac_real_imag); //resets ac_abs_phase mode to
    complex numbers

//close the old file (if it was given by name)
tr1->reopen(string name, ios_base::openmode mode=ios_base::out |
    ios_base::trunc); //… and open a new file with the optional mode
tr1->reopen(ostream* out_stream);  //… and replace the current out_stream

sca_trace(sctr, sca_elec_node, string signame);          // voltage tracing
sca_trace(sctr, module.sca_elec_port, string signame);   // voltage tracing

sca_trace(sctr,sca_sdf_signal, string signame) ;
sca_trace(sctr,module.sca_sdf_in, string signame) ;
sca_trace(sctr,module.sca_sca_sdf_out, string signame) ;
                :
```

Tracing SystemC-AMS Signals into a SystemC Tracefile

- sca_sdf_signals, sca_elec_node and the corresponding ports can be traced into a sc_trace_file
- Therefore overloaded sc_trace functions are provided
- This functions instantiate a converter module from a sca_sdf_signal/sca_elec_node to a sc_signal – the events of this internal sc_signal will be traced

```
sc_trace_file* sctr;
        sctr=sc_create_vcd_trace_file(string name); //Standard SystemC



sc_trace(sctr, sca_elec_node, string signame);              // voltage tracing
sc_trace(sctr, module.sca_elec_port, string signame);       // voltage tracing

sc_trace(sctr,sca_sdf_signal, string signame) ;
sc_trace(sctr,module.sca_sdf_in, string signame) ;
sc_trace(sctr,module.sca_sca_sdf_out, string signame) ;
```