SystemC-AMS
Analog & Mixed-Signal System Design
Alexander de Graaf, EEMCS-CAS
5/14/14
Outline

1. Introduction
2. Modeling Formalisms
3. Simulation and Tracing
4. Example: Bask Modulator
1.

Introduction
Motivation

- Interaction between HW/SW systems and their analog physical environment
- Leads to systems of digital HW/SW interwoven with analog and mixed signal blocks. Embedded Analog/Mixed-Signal (E-AMS) systems
SystemC suite

Functional

Architecture

Implementation

- Analog/Mixed-signal
  - VHDL-AMS
  - Verilog-AMS

- Digital
  - SystemVerilog
  - VHDL

NEW SystemC
AMS extensions

Specification

SoC

Interface

Dig

RE
Use cases

Use cases
- Executable specification
- Virtual prototyping
- Architecture exploration
- Integration validation

Model abstractions
- Discrete-time static non-linear
- Continuous-time dynamic linear
- Non-conservative behavior
- Conservative behavior

Modeling formalism
- Timed Data Flow (TDF)
- Linear Signal Flow (LSF)
- Electrical Linear Networks (ELN)
Language architecture

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

User features
- Classes and interfaces defined in the AMS language standard

Enabling technology
- Classes and interfaces not defined in the AMS language standard (implementation defined)

Semantics defined in the AMS language standard
- 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)
2.

Modeling Formalisms
1. Timed Data Flow (TDF)
2. Linear Signal Flow (LSF)
3. Electrical Linear Networks (ELN)
2.1

Timed Data Flow (TDF)
TDF Modeling Fundamentals

- Based on Synchronous Data Flow modeling formalism
- Considers data as signal sampled in time.
- Signals are
  - tagged at discrete points in time
  - Carry discrete or continuous values

![Diagram of TDF modules with arrows and labels](Image)
TDF Multi-Rates

Simulation sample time = 62.5\,\mu s

\[
\text{out\_sample\_freq} = \frac{\text{in\_sample\_freq}}{\text{out\_sample\_rate}} = \frac{\text{in\_sample\_freq}}{\text{in\_sample\_rate}}
\]
TDF Model Topologies

Model/Port attributes

Multi-rate

Loop with delay
TDF Module Construct

```c++
SCA_TDF_MODULE (my_tdf_module )
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<double> out;

  SCA_CTOR(my_tdf_module) {} // ! Constructor

  void set_attributes (){} // [] rate,tstep,delay

  void initialize() {} // [] state

  void processing () {} // ! behavior

  void ac_processing () {} // [] ac-behavior
};
```
TDF Module

- attributes
  - set rate, timestep, delay, timeoffset
- initialization
  - set initial values on ports, state, local variables
- processing
- local time
  - may be different in multirate models
  - use get time instead of sc_time_stamp
- constructor
  - SCA_CTOR
- usage constraints
  - A TDF module is primitive (so no instantiation of submodules)
  - Structural composition possible through regular SC_MODULE
  - SystemC Methods, Threads, wait, next_trigger, sensitive are not allowed!
TDF Ports

**SCA_TDF_MODULE** (my_tdf_module) {
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<double> out;
  sca_tdf::sca_de::sca_in<double> inp1;
  sca_tdf::sca_de::sca_in<double> inp2;
  // Rest of module
};

Port Attributes:
- set_timestep, get_time_step
- set_rate, get_rate
- set_delay, get_delay
- set_timeoffset, get_timeoffset

- Four classes of TDF ports
  - (input)   sca_tdf::sca_in<T>,
  - (output) sca_tdf::sca_out<T>
  - (converter input) sca_tdf::sca_de::sca_in<T>
  - (converter output) sca_tdf::sca_de::sca_out<T>
Modeling discrete/continuous-time behavior

Module time step (Tm)

TDF module
Instance of class
sca_tdf::sca_module

TDF output port
Instance of class
sca_tdf::sca_out<T>

sin_src
Tm:0.125ms

out
Discrete-time modeling

```cpp
SC_TDF_MODULE (sin_src) {
    sca_tdf::sca_out<double> out; // output port

    sin_src( sc_core::sc_module_name nm, double ampl_ = 1.0, double freq_ = 1.0e3,
             sca_core::sca_time Tm_ = sca_core::sca_time(0.125, sc_core::SC_MS))
        : out("out"), ampl(ampl_), freq(freq_), Tm(Tm_) {}

    void set_attributes() {
        set_timestep(Tm);
    }

    void processing() {
        double t = get_time().to_seconds(); // actual time
        out.write( ampl * std::sin( 2.0 * M_PI * freq * t ) );
    }

private:
    double ampl; // amplitude
    double freq; // frequency
    sca_core::sca_time Tm; // module time step
};
```
Continuous-time behavior

TDF primitive module embedding a Laplace transfer function

\[ H(s) = \frac{H_0}{1 + \frac{1}{2\pi f_c} s} \]

Low-pass filter: \( H_0 \) is DC-gain, \( f_c \) is cut-off frequency
Continuous-time modeling (sca_ltf_nd)

```
SC_TDF_MODULE (ltf_nd_filter) {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;

    double fc;  // 3dB cut-off frequency in Hz
    double h0; // DC gain

    SCA_CTOR(ltf_nd_filter) : in("in"), out("out"), fc(1.0e3), h0(2.0) {}

    void initialize() {
        num[0] = 1.0;
        den[0] = 1.0;
        den[1] = 1.0 / (2.0 * M_PI * fc);
    }

    void processing() {
        out.write( ltf_nd(num, den, in.read(), h0) );
    }

    private:
        sca_tdf::sca_ltf_nd ltf_nd; // Laplace transfer function
        sca_util::sca_vector<double> num, den; // numerator and denominator coefficients
};
```

\[
H(s) = k \cdot \sum_{i=0}^{M-1} \frac{num_i \cdot s^i}{\sum_{i=0}^{N-1} den_i \cdot s^i} \cdot e^{-s \cdot delay}
\]
Continuous-time modeling (sca_ltf_zp)

```cpp
SC_TDF_MODULE (ltf_zp_filter) {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;

    double fc; // 3dB cut-off frequency in Hz
    double h0; // DC gain

    SCACTOR(ltf_zp_filter) : in("in"), out("out"), fc(1.0e3), h0(2.0) {}

    void initialize() {
        // filter requires no zeros to be defined
        poles(0) = sca_util::sca_complex( -2.0 * M_PI * fc, 0.0 );
        k = h0 * 2.0 * M_PI * fc;
    }

    void processing() {
        out.write( ltf_zp( zeros, poles, in.read(), k ) );
    }

    private:
    double k; // filter gain
    sca_tdf::sca_ltf_zp ltf_zp; // Laplace transfer function
    sca_util::sca_vector<sca_util::sca_complex > poles, zeros; // poles and zeros as complex values
};
```

The transfer function is given by:

\[
H(s) = k \cdot \frac{\prod_{i=0}^{M-1} (s - z_{eros_i})}{\prod_{i=0}^{N-1} (s - p_{oles_i})} \cdot e^{(-s \cdot delay)}
\]
Continuous-time modeling (sca_ss)

```cpp
SC_TDF_MODULE (ss_filter) {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;

double fc; // 3dB cut-off frequency in Hz
SCA_CTOR(ss_filter) : in("in"), out("out"), fc(1.0e3) {}  

void initialize() {
    double r_val = 1e3;
    double c_val = 1.0 / ( 2.0 * M_PI * fc * r_val);

    a(0,0) = -1.0 / ( c_val * r_val );
    b(0,0) = 1.0 / r_val;
    c(0,0) = 1.0 / c_val;
    d(0,0) = 0.0;
}

void processing() {
    sca_util::sca_vector<double> x;  x(0) = in.read();
    sca_util::sca_vector<double> y = state_space1( a, b, c, d, s, x );    out.write(y(0));
}

private:
    sca_tdf::sca_ss state_space1; // state-space equation
    sca_util::sca_matrix<double> a, b, c, d; // state-space matrices
    sca_util::sca_vector<double> s; // state vector ;
```

\[
\frac{ds(t)}{dt} = A \cdot s(t) + B \cdot x(t - \text{delay}) \\
y(t) = C \cdot s(t) + D \cdot x(t - \text{delay})
\]
Structural composition of TDF modules

Discrete-event input port
Instance of class sc_core::sc_in<T>

TDF input converter port
Instance of class sca_tdf::sca_de::sca_in<T>

Port-to-port binding

Discrete-event signal
Instance of class sc_core::sca_signal<T>

Port-to-port binding

TDF output converter port
Instance of class sca_tdf::sca_de::sca_out<T>

TU Delft
Interaction TDF en DE domain

- Reading from the DE domain

- Writing to the DE domain
TDF execution semantics

- **TDF module attribute settings:**
  Execute all `set_attributes` member functions

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

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

- **TDF module initialization:**
  Execute all `initialized` member functions once

- **TDF module activation and processing:**
  Repeatedly execute all processing member functions

- **TDF module post processing:**
  Execute all `end_of_simulation` member functions once

---

**elaboration phase**

**simulation phase**
2.2

*Linear Signal Flow (LSF)*
LSF Model of Computation

- Model behavior as relations between variables of a set linear algebraic equations
- Continuous time modeling style
- One real-value quantity for each signal
- Typically blockdiagrams composed of a finite set of predefined LSF modules like adders, multipliers, integrators etc. (! No user defined modules)
- Only input ports, output ports (No inout ports)
Setup of LSF equation system

\[ y(t) = k_1 \cdot \frac{dx(t)}{dt} + k_2 \cdot \frac{dy(t)}{dt} \]

LSF primitives:

- **a)** Weighted addition (add)
  \[ y(t) = k_1 x_1(t) + k_2 x_2(t) \]
- **b)** Weighted subtraction (sub)
  \[ y(t) = k_1 x_1(t) - k_2 x_2(t) \]
- **c)** Multiplication (gain)
  \[ y(t) = k x(t) \]
- **d)** Scaled first-order time derivative (dot)
  \[ y(t) = k \frac{dx(t)}{dt} \]
LSF Language constructs

- A set of primitive modules:
  - `sca_lsf::sca_add, sub, gain, dot, integ, delay` etc.
- Module time_step has to be assigned:
  - by member function `set_timestep`
  - through propagation in LSF cluster
  - through connection with a TDF cluster
- Ports:
  - Basic I/O: `sca_lsf::sca_in, out`
  - Convert to DE: `sca_lsf::sca_de::sca_source, sink`
  - Convert to TDF: `sca_lsf::sca_tdf::sca_source, sink`
- Signals to connect primitive modules: `sca_lsf::sca_signal`
**LSF model encapsulation**

```
SC_MODULE (lsf_in_tdf) {
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;

    sca_lsf::sca_add add1;
    sca_lsf::sca_dot dot1;
    sca_lsf::sca_gain gain1;
    sca_lsf::sca_tdf::sca_source tdf2lsf;
    sca_lsf::sca_tdf::sca_sink lsf2tdf;

    lsf_in_tdf( sc_core::sc_module_name, double k, double k2 )
        : in("in"), out("out"), sig1("sig1"), sig2("sig2"), sig3("sig3"), sig4("sig4"),
          add1("add1"), dot1("dot1", k), gain1("gain1", k2), tdf2lsf("tdf2lsf"), lsf2tdf("lsf2tdf") {
            tdf2lsf.inp(in);
            tdf2lsf.y(sig1);
            add1.x1(sig1);
            add1.x2(sig3);
            add1.y(sig2);
            dot1.x(sig2);
            dot1.y(sig4);
            gain1.x(sig4);
            gain1.y(sig3);
            lsf2tdf.x(sig4);
            lsf2tdf.outp(out);
        }
};
```
LSF execution semantics

LSF time step calculation and propagation:
Define time step and check consistency

LSF equation set-up and solvability check:
Define the equation system and check if it can be solved

LSF initialization:
Set initial conditions, e.g., defined in LSF primitives

LSF time-domain simulation:
Provide results at the calculated time points
2.3

*Electrical Linear Networks (ELN)*
ELN Model of Computation

• Model behavior as relations between variables of a set linear algebraic equations

• Conservative, continuous time modeling style

• Voltage and current quantities follow KVL and KCL.

• Typically a network composed of a finite set of predefined primitive modules (R,L,C, vsources, isources etc) interconnected by ELN nodes. (Spice like description)

• Module terminals serve to interconnect with other ELN modules
Setup of ELN equation system

**ELN equations**

\[-i_1 + \frac{v_a}{R_1} + C \cdot \frac{d(v_{a,b} + q^0)}{C} dt = 0\]

\[\frac{v_b}{R_2} - C \cdot \frac{d(v_{a,b} + q^0)}{dt} = 0\]

**basic ELN lumped elements**

- For resistor: \[v_{p,n}(t) = i_{p,n}(t) \cdot R\]
- For capacitor: \[i_{p,n}(t) = C \cdot \frac{d(v_{p,n}(t) + q^0)}{dt}\]
- For inductor: \[v_{p,n}(t) = L \cdot \frac{d(i_{p,n}(t) + \phi_{0})}{dt}\]
ELN Language constructs

- A set of primitive modules:
  - `sca_eln::sca_r, l, c, v cvs, c cvs, v ccs, c ccs` etc.
- Module time_step has to be assigned:
  - by member function set_timestep
  - through propagation in ELN cluster
  - through connection with a TDF cluster
- Ports:
  - Basic I/O: `sca_eln::sca_terminal`
  - Convert to DE: `sca_eln::sca_de::sca_[vi]source, [vi]sink`
  - Convert to TDF: `sca_eln::sca_tdf::sca_[vi]source,[vi] sink`
- Signals to connect primitive modules:
  - `sca_eln::sca_node`
  - `sca_eln::sca_node_ref (gnd)`
Interaction between ELN and DE

Discrete-event signal
Instance of class
sc_core::sc_signal<double>

ELN converter module
Instance of class
sca_eln::sca_de::sca_vsource

ELN output voltage
ELN converter module
Instance of class
sca_eln::sca_de::sca_isource

ELN input voltage
ELN converter module
Instance of class
sca_eln::sca_de::sca_vsink

Discrete-event signal
Instance of class
sc_core::sc_signal<double> or
sc_core::sc_buffer<double>

ELN converter module
Instance of class
sca_eln::sca_de::sca_isink
Interaction between ELN and TDF
ELN model encapsulation

```cpp
SC_MODULE(eln_in_tdf) {
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<double> out;
  sca_eln::sca_tdf::sca_vsource vin;
  sca_eln::sca_tdf::sca_vsink vout;
  sca_eln::sca_r r;
  sca_eln::sca_c c;
  eln_in_tdf( sc_core::sc_module_name, double r_val, double c_val )
    : in("in"), out("out"), n1("n1"), n2("n2"), vin("vin"), vout("vout"), r("r", r_val), c("c", c_val)
    {
      vin.inp(in);
      vin.p(n1);
      vin.n(gnd);
      r.p(n1);
      r.n(n2);
      c.p(n2);
      c.n(gnd);
      vout.p(n2);
      vout.n(gnd);
      vout.outp(out);
    }
  private:
    sca_eln::sca_node n1, n2;
    sca_eln::sca_node_ref gnd;
};
```
ELN execution semantics

ELN time step calculation and propagation:
Define time step and check consistency

ELN equation set-up and solvability check:
Define the equation system and check if it can be solved

ELN initialization:
Set initial conditions, e.g., defined in ELN primitives

ELN time-domain simulation:
Provide results at the calculated time points
3.

*Simulation and Tracing*
#include <systemc-ams>
#include "my_source.h"
#include "my_control.h"
#include "my_dut.h"
#include "my_sink.h"

• use smallest resolution possible (1 fs) allows $2^{64}$ fs (approx. 5 hours)

int sc_main(int argc, char* argv[]) {
    sc_core::sc_set_time_resolution (1.0, sc_core::SC_FS);
    sca_tdf::sca_signal <double> sig1, sig2;
    sc_core::sc_signal <bool> sc_sig;

    my_source i_my_source("i_my_source");
    i_my_source.out(sig1);

    my_control i_my_ctrl("i_my_ctrl");
    i_my_ctrl.out(sc_sig);

    my_dut i_my_dut("i_my_dut");
    i_my_dut.in(sig1);
    i_my_dut.ctrl(sc_sig);
    i_my_dut.out(sig2);

    my_sink i_my_sink("i_my_sink");
    i_my_sink.in(sig2);

    sc_core::sc_start (10, sc_core::SC_MS);
    return 0;
}
Tracing

- Record simulation waveform results into trace files
- Trace of AMS signals, nodes, ports, terminals or variables
- Formats:
  - VCD format only for time-domain simulations
  ```
  // open trace file in VCD format
  sca_util::sca_trace_file* atf = sca_util::sca_create_vcd_trace_file( "my_trace.vcd" );
  ...
  // close the trace file
  sca_util::sca_close_vcd_trace_file( atf );
  ```
  - Tabular format for time/frequency-domain simulations
    control functions: enable, disable, reopen set_mode
  ```
  // open trace file in tabular format
  sca_util::sca_trace_file* atf = sca_util::sca_create_tabular_trace_file( "my_trace.dat" );
  ... // trace in tabular format to the shell
  ... sca_util::sca_trace_file* atfs = sca_util::sca_create_tabular_trace_file( std::cout );
  // close the trace file
  sca_util::sca_close_tabular_trace_file( atf );
  ```
  output to file output to stream
Tracing AMS supported signals

- TDF: TDF ports, TDF signals, variables derived from `sca_util::sca_trace_variable`
- LSF: LSF ports and LSF signals
- ELN: voltage tracing on node and terminals
- SystemC: ports and signals

```cpp
sca_util::sca_trace_file* tf = sca_util::sca_create_tabular_trace_file("trace.dat");  

sca_util::sca_trace(tf, sig1, "sig1");  
sca_util::sca_trace(tf, sig2, "sig2");  

sc_core::sc_start(2, sc_core::SC_MS);  

tf->reopen("ac_trace.dat");  

tf->set_mode(sca_util::sca_ac_format(sca_util::SCA_AC_MAG_RAD));  

sca_ac_analysis::sca_ac_start(1.0e3, 1.0e6, 4, sca_ac_analysis::SCA_LOG);  

sca_util::sca_close_tabular_trace_file(tf);  
```
Small-signal frequency-domain simulation

- Start in sc_main using:
  - sca_ac_analysis::sca_ac_start
  - sca_ac_analysis::sca_ac_noise_start
- Call frequency/time – domain start functions in any order

```c++
// frequency-domain simulations from 1kHz to 10KHz with 100 points on a linear scale:
sca_ac_analysis::sca_ac_start(1e3, 10e3, 100, sca_ac_analysis::SCA_LIN);
sca_ac_analysis::sca_ac_noise_start(1e3, 10e3, 100, sca_ac_analysis::SCA_LIN);

// frequency-domain simulations from 1Hz to 1MHz with 1001 points on a logarithmic scale:
sca_ac_analysis::sca_ac_start(1.0, 1e6, 1001, sca_ac_analysis::SCA_LOG);
sca_ac_analysis::sca_ac_noise_start(1.0, 1e6, 1001, sca_ac_analysis::SCA_LOG);
```
4.

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

SCA_TDF_MODULE(sine) {
    sca_tdf::sca_out<double> out; // output port
    void processing(){
        out.write( sin(sc_time_stamp().to_seconds()*(1000.*2.*M_PI)));
    }
    SCACTOR(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

```cpp
#include "systemc-ams.h"

SCA_TDF_MODULE(drain) { // a drain module to connect the signal to
    sca_tdf::sca_in<double> in; // input port
    SCA_CTOR(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 \textit{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 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

```c
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;
  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 de-modulation](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

```
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

```cpp
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

```c
SCA_TDF_MODULE(bit_recov){
    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

```
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;}

Instantiating and connecting
• 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

```c
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.  

![Waveform diagram showing signal transitions over time](image)
A simple environment model

```
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
Thank you for your attention!

Your:
• questions
• comments
• ideas
• objections