ESL Design and Modeling with SystemC AMS for Mixed-Signal IoT and Automotive Applications

Karsten Einwich - COSEDA Technologies
Martin Barnasconi, Sumit Adhikari - NXP Semiconductors
Christoph Grimm, Carna Radojicic - TU Kaiserslautern
Torsten Mähne - Berner Fachhochschule

accellera SYSTEMS INITIATIVE

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Outline

- Introduction, requirements & use cases
- SystemC AMS Methodology
- Introduction to the SystemC AMS design language
- SystemC AMS Modeling Techniques
- Design flow Integration and Tools
- Industrial application

INTRODUCTION, REQUIREMENTS & USE CASES

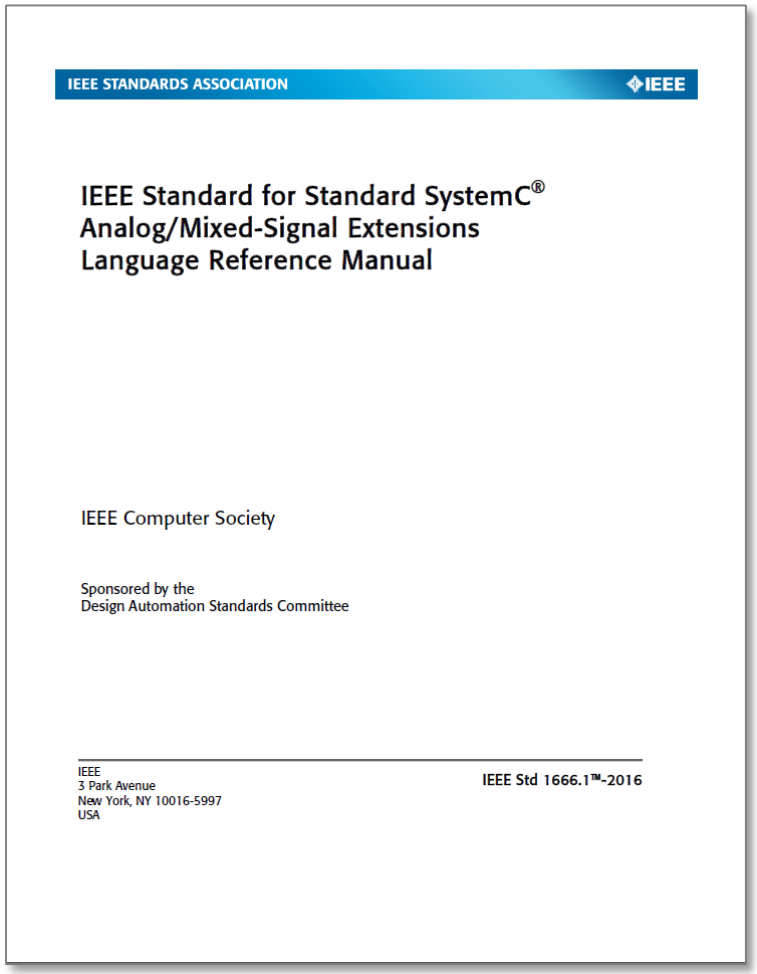
Introduction

- Most virtual prototypes only focus on digital HW/SW system components – AMS functionality is often neglected
- However, today's embedded systems contain AMS and RF components which tightly interact with the HW/SW system
- Advanced modeling approaches needed to include AMS behavior in the architecture design phase
- Application of a model-based ESL design refinement flow based on SystemC and SystemC AMS

Why SystemC AMS extensions?

- Unified and standardized modeling language to describe embedded mixed-signal architectures
  - **Abstract AMS model descriptions** supporting a design refinement methodology, from functional/algorithm down to implementation views
  - Enabling **tool-independent exchange and reuse** of AMS models and building blocks
  - System-level language for analog *and* digital signal processing functionality
- Facilitate the creation of mixed-signal virtual prototypes
  - Integration of abstract AMS/RF subsystems in combination with digital HW/SW subsystems
- Enrich ESL design and verification eco-system with system-level design tools and flows based on SystemC standards
  - IEEE Std 1666-2011 (SystemC LRM)
  - IEEE Std 1666.1-2016 (SystemC AMS LRM)

SystemC AMS standard (IEEE 1666.1-2016)


IEEE STANDARDS ASSOCIATION IEEE

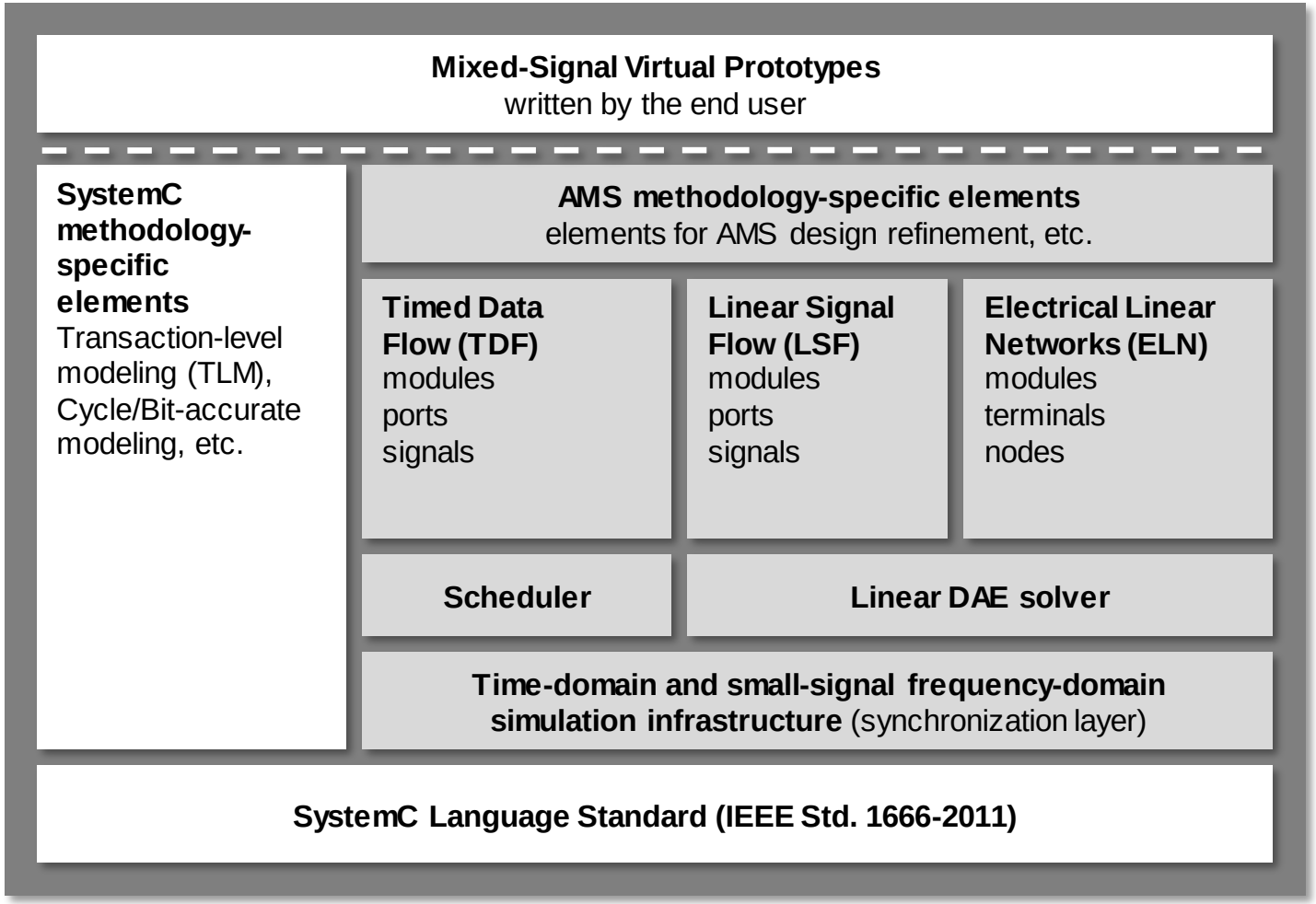
IEEE Standard for Standard SystemC®
Analog/Mixed-Signal Extensions
Language Reference Manual

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

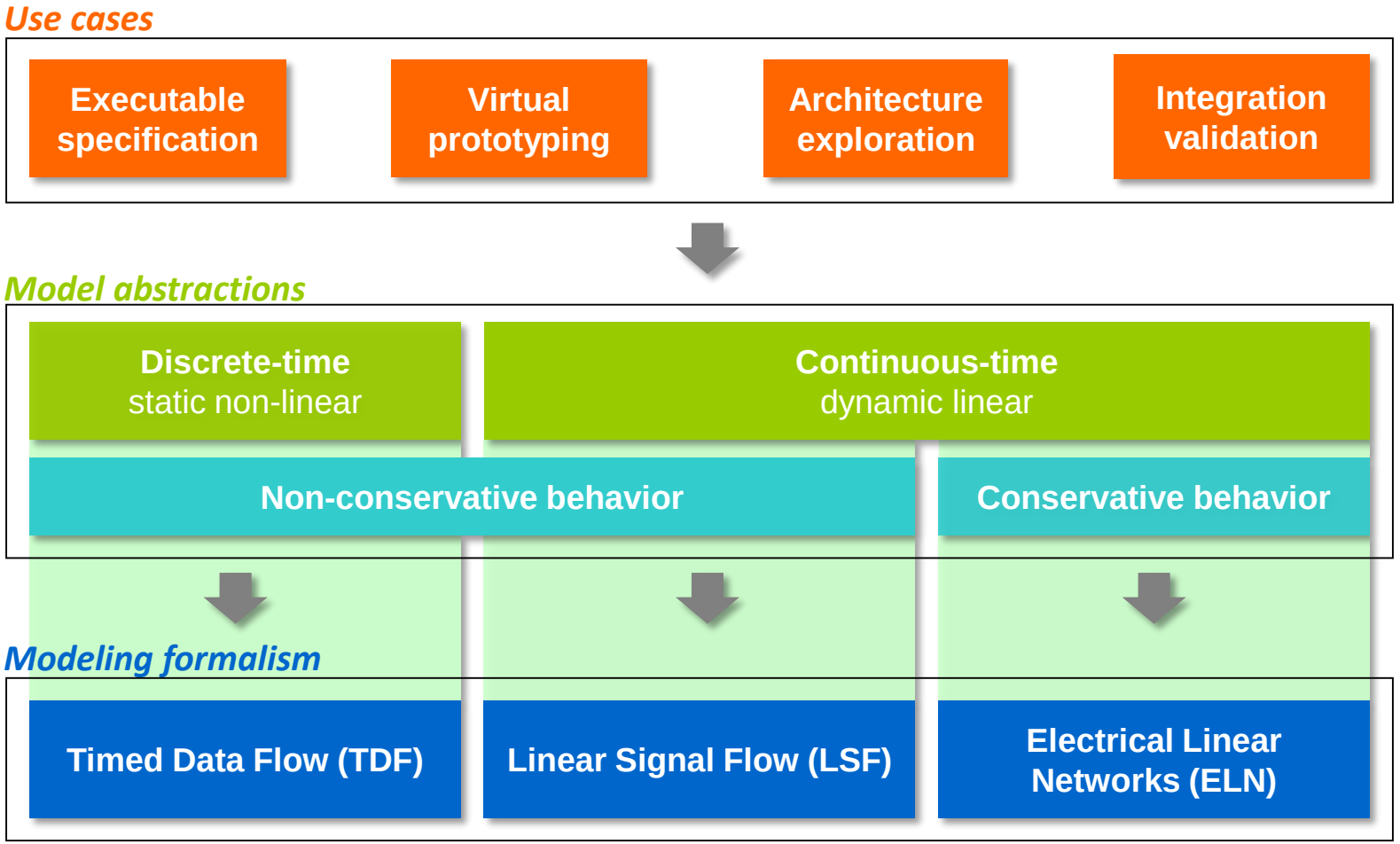
IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1666.1™-2016



Download free of charge thanks to Accellera sponsored IEEE Get Program:
https://standards.ieee.org/findstds/standard/1666.1-2016.html

SystemC AMS model abstractions and modeling formalisms



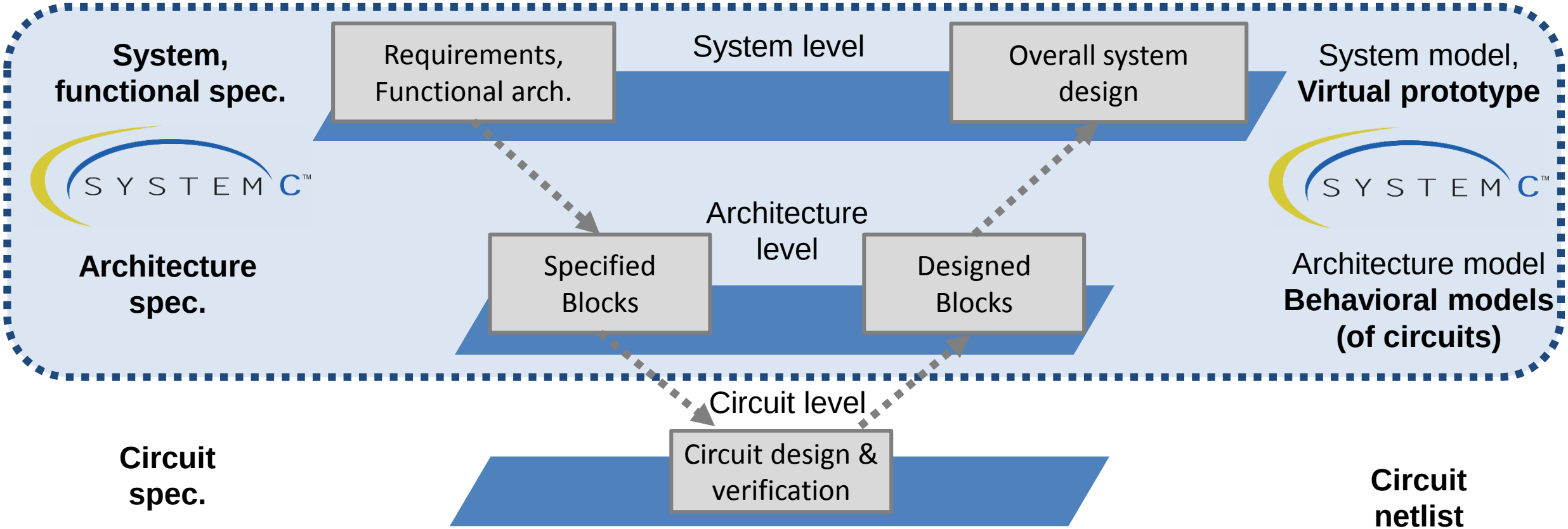
Source: Accellera Systems Initiative

SYSTEMC AMS: METHODOLOGY

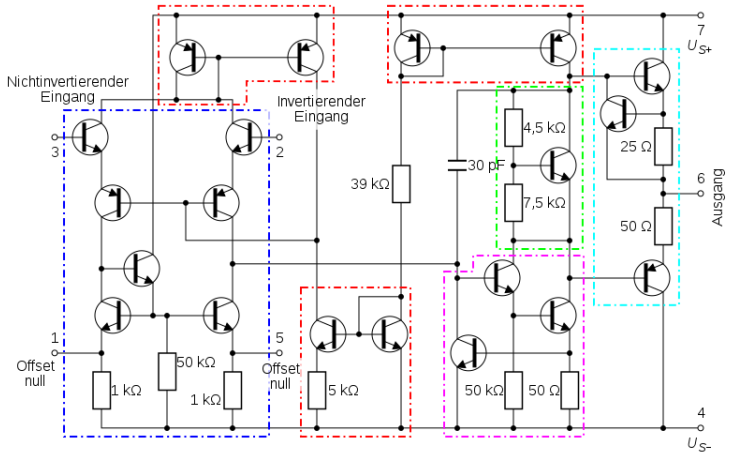
SystemC AMS: Methodology

- SystemC AMS Models of Computation
- Hierarchical Verification Flow
- Case Study

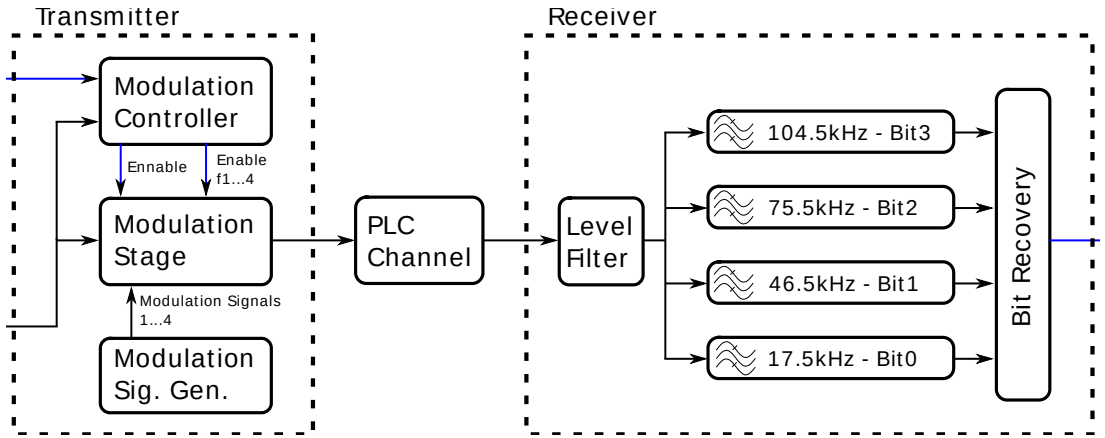
Hierarchical verification flow with SystemC (AMS)



SystemC AMS: Block Diagram Level



Circuit schematic, e.g. SPICE



Block diagram, e.g. Matlab/Simulink, SystemC AMS

SystemC AMS allows us verification&validation of AMS systems at **functional** and **block diagram level**

- **Directed interaction** of functional blocks
- Blocks are either **ideal functions** or **behavioral models** of circuits, no circuit-level models
- Behavior of blocks and semantics specified by SystemC AMS

Specification of a Block's Function

By a transfer function
- E.g. Filter:

```
y = ltf_1(nom, denom, y);
```

By a (static) function
- E.g. Mixer:

```
y = rf_in*carrier;
```

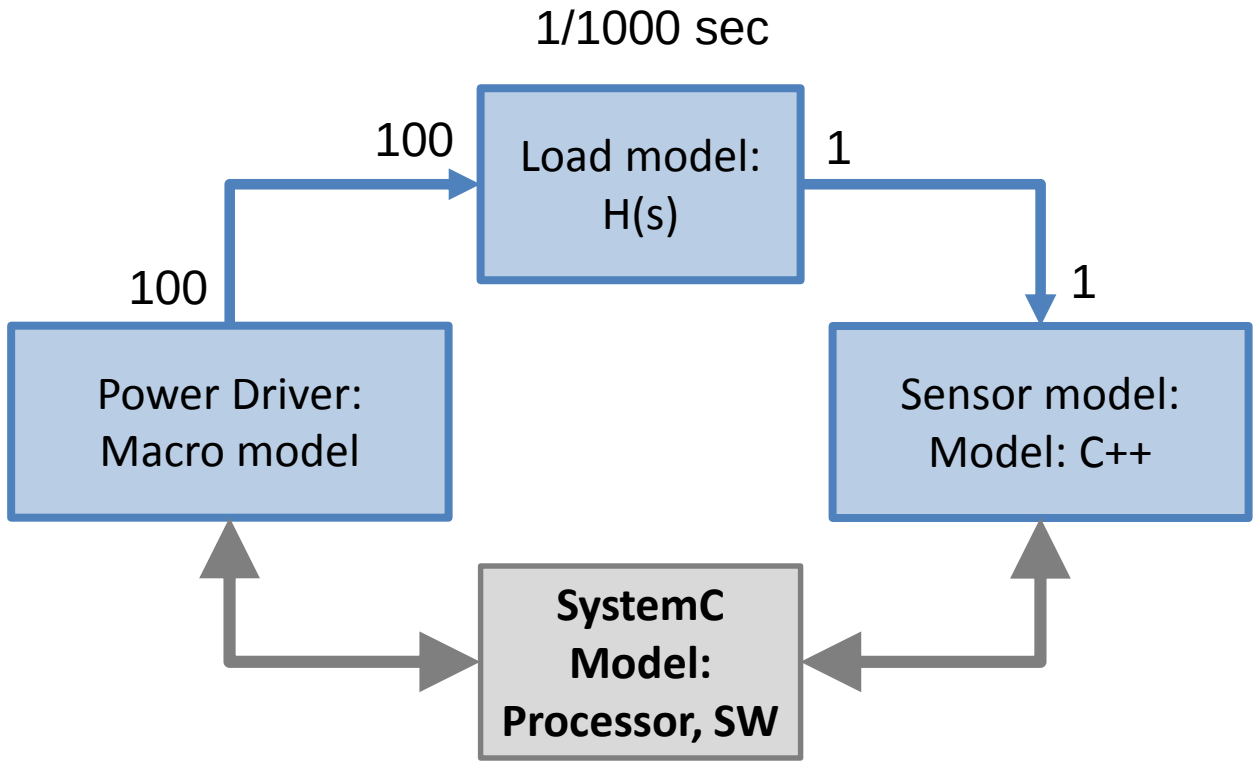
By a Macro Model
- E.g. power driver

```
(Macromodel: Switches, R L C)
```

… or by arbitrary C++ code, maybe mixing all the above options.

Timed Data Flow (TDF) and Linear Signal Flow (LSF)

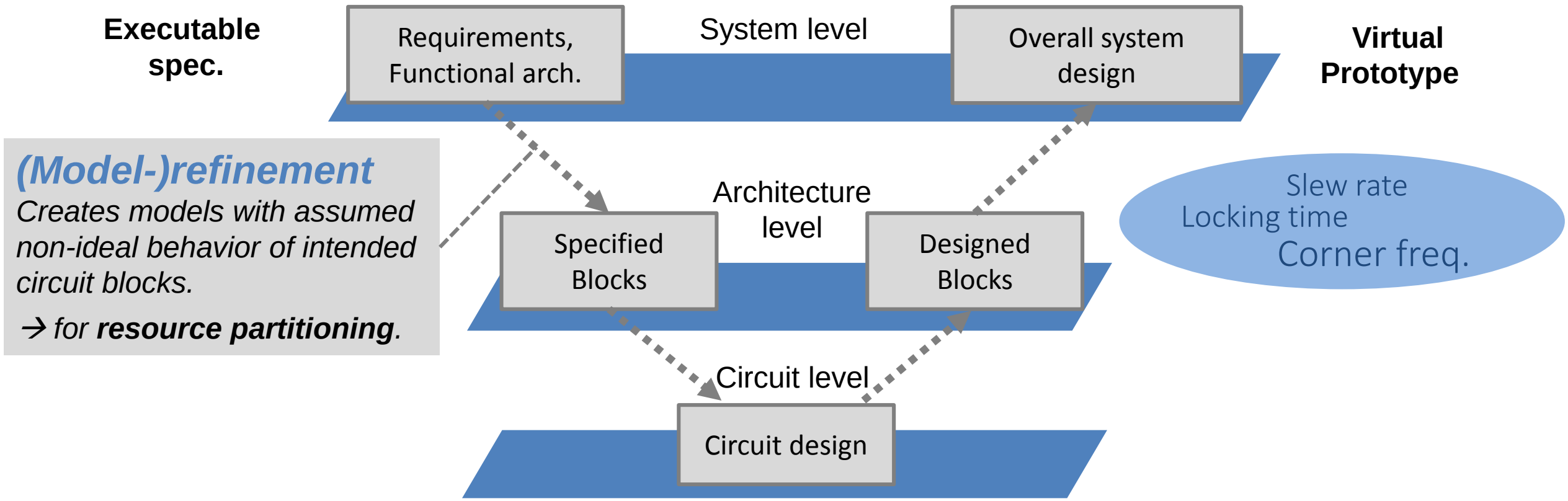
- Functions of blocks in block diagrams are processed **in data flow's direction**.
- Ports may have different rates
  - Static data-flow model
  - **Scheduling before simulation**
- Time steps are assigned to
  - Processing of block's functions
  - Distances between samples
- Time steps and rates specified in model
  - Dynamic re-scheduling during simulation (2.0)



SystemC AMS: Methodology

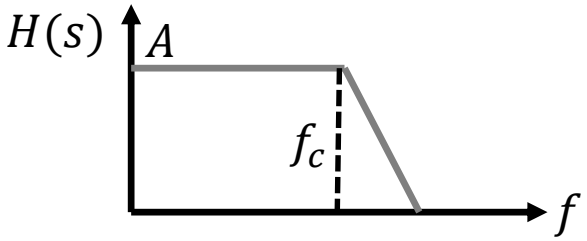
- SystemC AMS Models of Computation
- Hierarchical Verification Flow
- Case Study

Hierarchical Verification Flow: Model Refinement



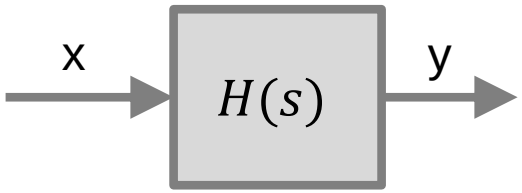
Simple Example: Executable Specification of a Filter

The function of a "filter" is described by a **transfer function H(s), e.g.**



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

SystemC AMS allows us to specify the behavior of a block by some C-Code (more later)




```
A(0) = 1.0;
denom(0)=1.0;
denom(1)=r2pi/fc;

y = ltf_block(A, denom, x);
```

Model Refinement of a Filter

Which are **relevant properties** of filter for the system?

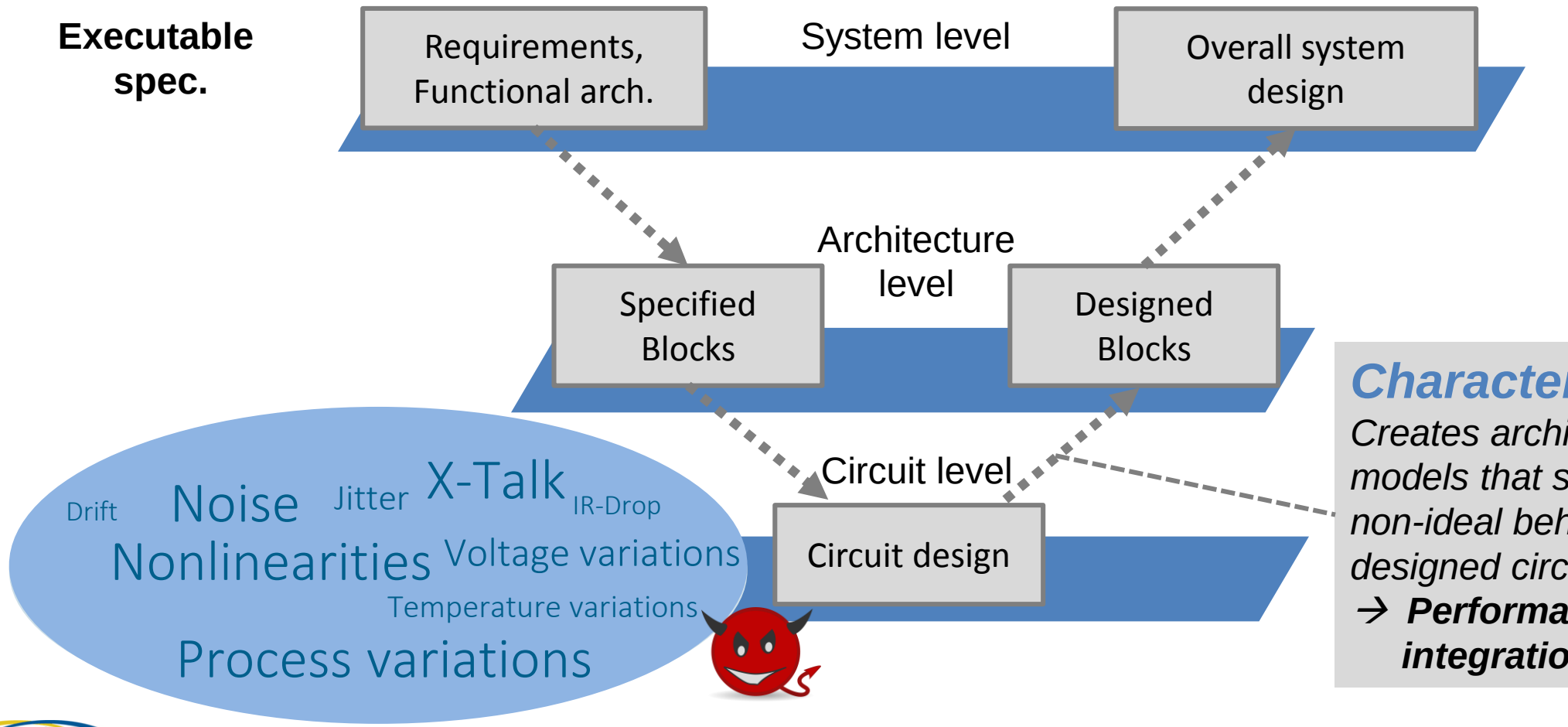
- Noise, limitation
- Accuracy of corner frequency f_c, A
- Non-linearity, slew-rate, limits at internal states, …

Objective: Resource Partitioning

- Distribute resources (i.e. accuracy) to blocks;
- Verify system performance prior to circuit design

```
A(0) = 1.0 + dA;              // Range of possible A
denom(0)=1.0;
denom(1)=r2pi/(fc + df);      // Range of possible fc

if (x > 5.0) then x = 5.0; // limitation at input
x += noise(3);             // assume some noise

y = ltf_block(A, denom, x);

if y > 5.0 then y = 5.0;   // limitation at output
```

Hierarchical Verification Flow: Characterization



Characterization

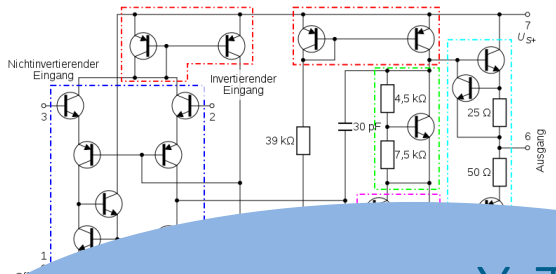
Creates architecture-level models that show the relevant non-ideal behavior of a designed circuit.

→ ***Performance verification**; **integration validation**.*

Characterization for System Verification

Circuit designers **use accurate, appropriate tools** for circuit level design & verification

- Example: SPICE; we don't recommend to change that!
- Use circuit simulator & models for **characterization of blocks**!



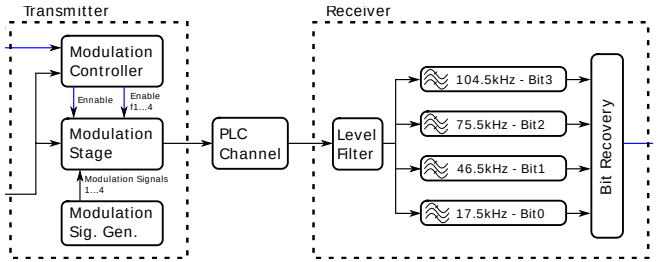
Drift Noise Jitter X-Talk IR-Drop
Nonlinearities Voltage variations
Temperature variations
Process variations

Corners of circuit parameters.

Slew rate
Locking time noise
Fc, A

Worst-case properties <u>of block</u>.

Characterization plan; w/ e.g. SPICE simulations



Verify system performances

e.g. BER

System integration w/ SystemC AMS

Simple Example: Characterization of Filter

Characterization plan:
With circuit-simulator, do:

- AC Analysis → Ranges for A, fc
- DC Sweep → Limitation of output
- non-linearity, slew-rate, noise, ...

```
A(0) = 1.0 + dA;              // Range of possible A
denom(0)=1.0;
denom(1)=r2pi/(fc + df);      // Range of possible fc
if (x > 5.0) then x = 5.0; // limitation at input
x += noise(3);                // assume some noise
y = ltf_block(A, denom, x);
if y > 5.0 then y = 5.0;      // limitation at output
```

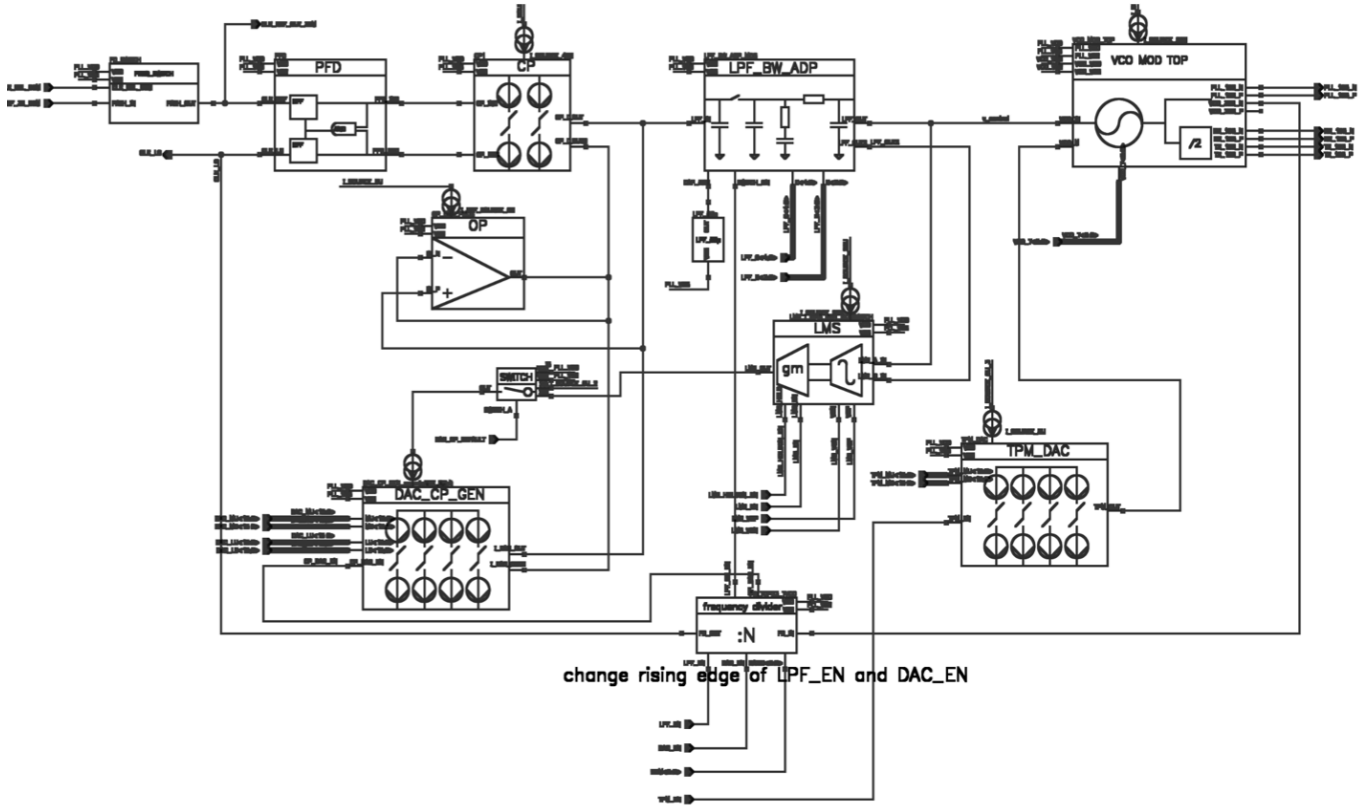
Verification of system integration & performance:

- Build SystemC (AMS) model with the above model
- Determine performances according to the verification plan
- Take care of correlations! ! !
  (See Grimm, Rathmair: *"Dealing with Uncertainties in Analog/Mixed-Signal Systems" Proceedings Design Automation Conference,* DAC 2017)

SystemC AMS: Methodology

- SystemC AMS Models of Computation
- Hierarchical Verification Flow
- Case Study

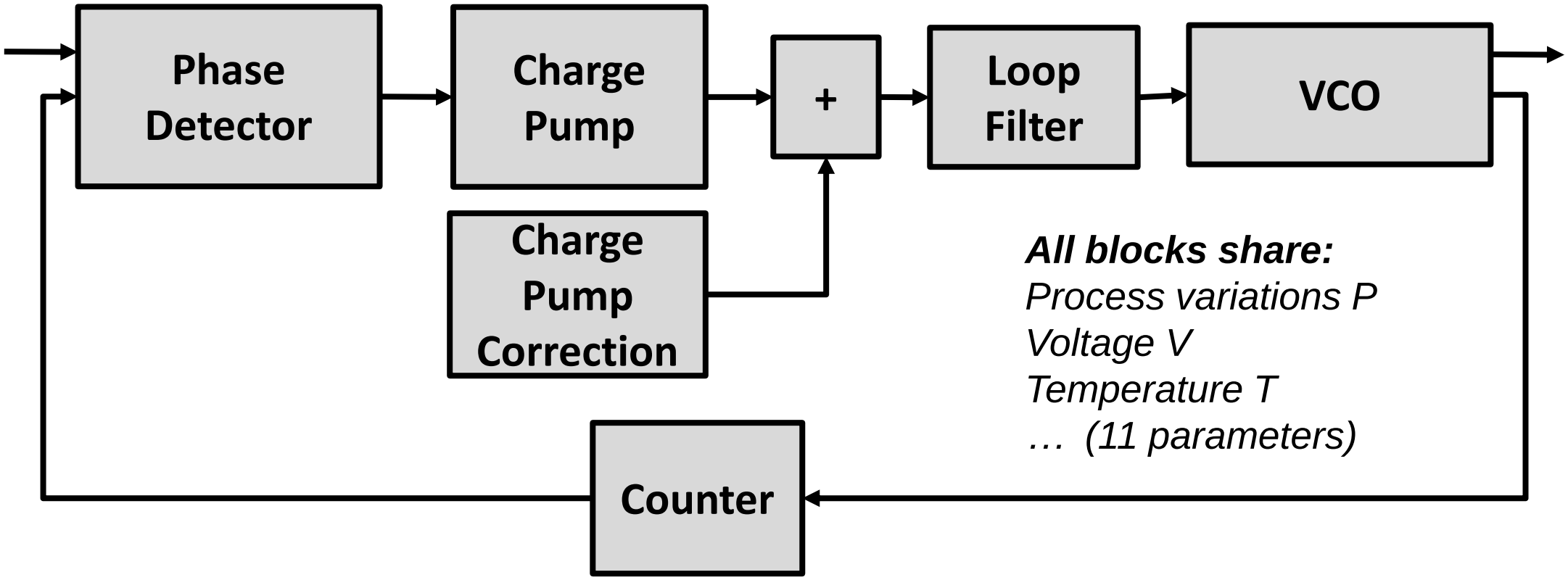
Example for Hierarchical Verification of a PLL of an IEEE 802.15.4 RF transceiver



Chip designed with Cadence, exported as SystemC AMS model.
7-12 uncertainties (PVT, …) in various parts, partially correlated.

- **Characterization** using ELDO models (Mentor)
- **System simulation** with Design-of-Experiments (DoE) to find estimation of Worst Case performance.
- **Symbolic simulation**

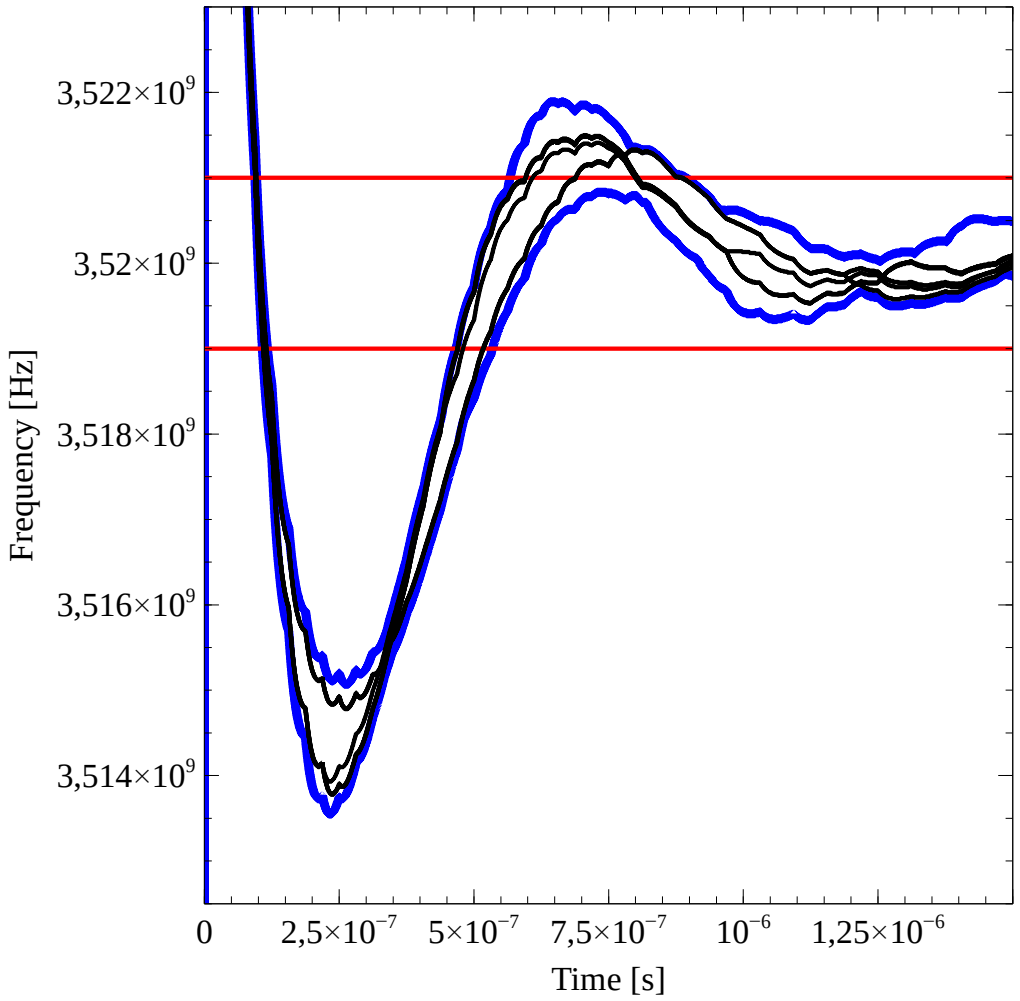
SystemC AMS Model, Block Diagram Level



Simulation with WC simulation & DoE

Black:
WC by DoE

Blue:
Symbolic
WC analysis



PLL simulated for 1.5 µs;
sampling frequency of 50 GHz
(75.000 time steps);
12 uncertainties.

- Single simulation run **1-2 sec**
- **DoE simulation and symbolic simulation to find Worst Case corners (some minutes)**

Summary Methodology

SystemC AMS

- Is made for fast and efficient architecture-level simulations
- Circuits are modeled by blocks or simple macro-models

Refinement of executable specification

- Allows us to find bottlenecks, estimates feasibility prior to circuit level design
- Reduces risk of failing projects

Characterization

- After circuit design; matches behavioral models with “real” circuit level models

Part 3

INTRODUCTION TO THE SYSTEMC AMS DESIGN LANGUAGE

SystemC AMS Language Basics

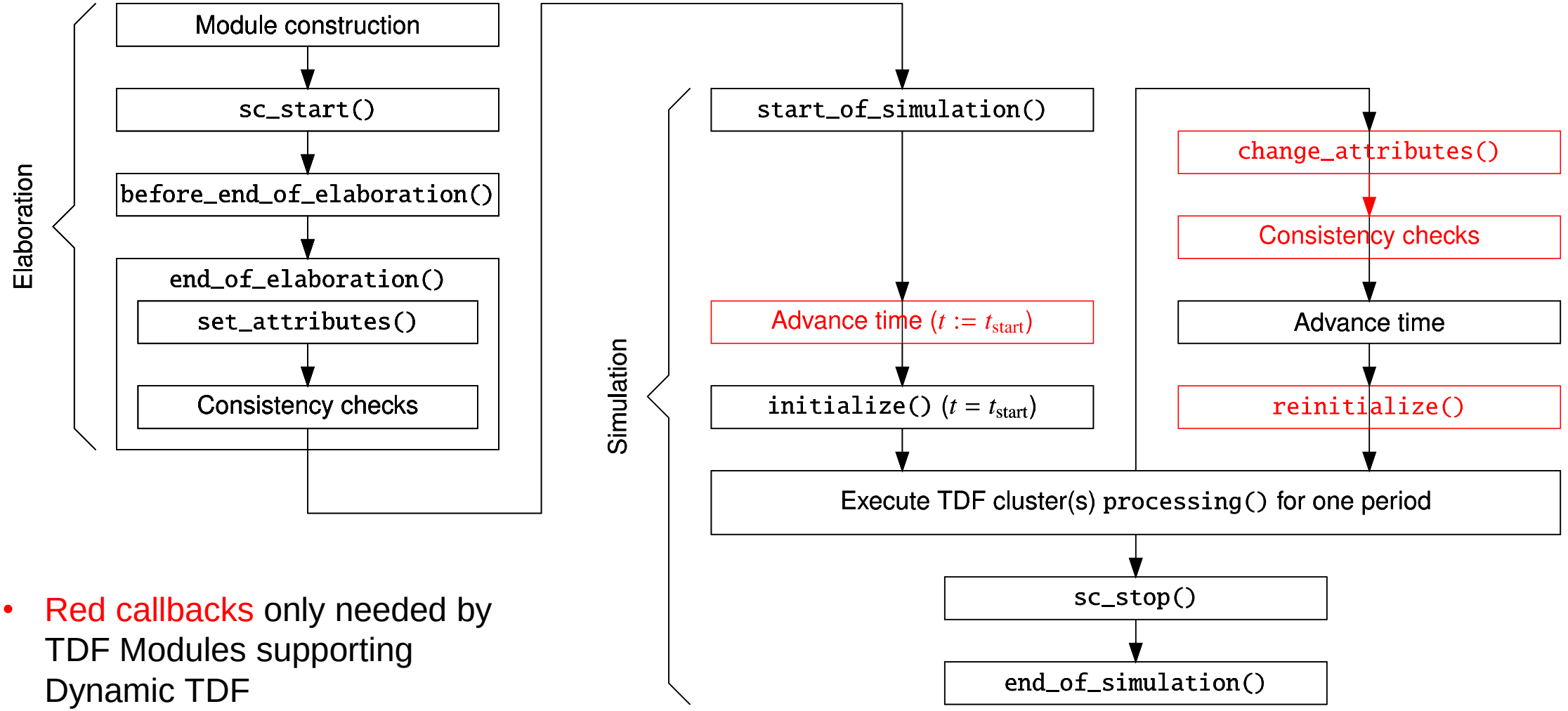
- Basic keywords defined as classes in the LRM and header `<systemc-ams>`:
  - **`sca_module`** – base class for SystemC AMS primitive
  - **`sca_in`** / **`sca_out`** – non-conservative (directed in / out port)
  - **`sca_terminal`** – conservative terminal
  - **`sca_signal`** – non-conservative (directed) signal
  - **`sca_node`** / **`sca_node_ref`** – conservative node / ground reference
- The model of computation is assigned by the namespace, e.g.:
  - **`sca_tdf::sca_module`** – base class for timed data flow modules
  - **`sca_lsf::sca_in`** – a linear signal flow input port
  - **`sca_tdf:sca_in<T>`** – a TDF input port of type **`T`**
  - **`sca_eln::sca_terminal`** – an electrical linear network terminal
  - **`sca_eln::sca_node`** – an electrical linear network node

TDF Module Structure

```
#include <systemc-ams>

class my_tdf_module : public sca_tdf::sca_module
{
public:
  sca_tdf::sca_in<Ti> tdf_in_i;
  sca_tdf::sca_out<Tj> tdf_out_j;
  sca_tdf::sca_de::sca_in<Tk> de_in_k;
  sca_tdf::sca_de::sca_out<Tl> de_out_l;
  // ...
  explicit my_tdf_module(sc_core::sc_module_name nm, /* ... */)
  : tdf_in_i("tdf_in_i"), tdf_out_j("tdf_out_j"), /* ... */
  { /* ... */ }

  void set_attributes() { /* ... */ }
  void initialize() { /* ... */ }
  void change_attributes() { /* ... */ }
  void reinitialize() { /* ... */ }
  void processing() { /* ... */ }
private:
  // ...
};  // class my_tdf_module
```

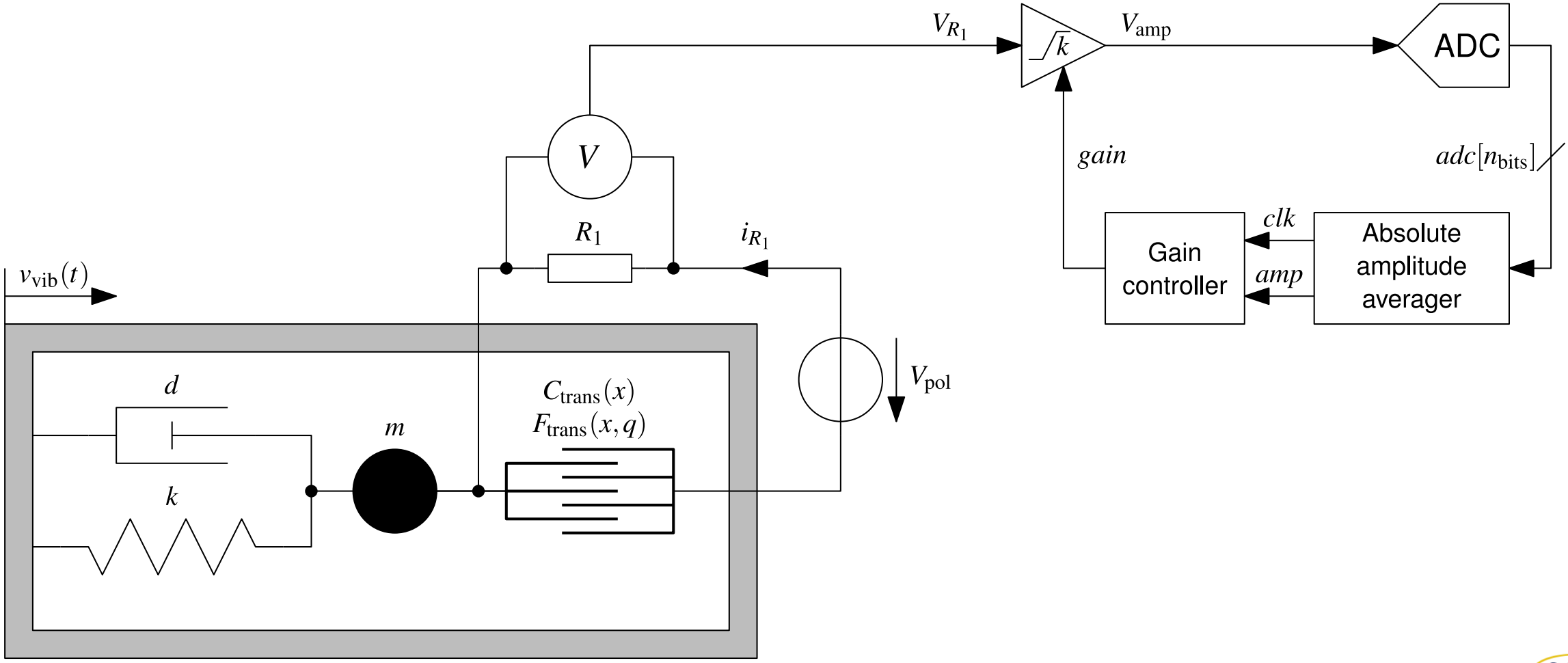
- TDF input & output ports
- DE⇄TDF input & output converter ports
- TDF module constructor for initialization of all ports and member variables
- TDF elaboration & simulation callbacks
- private state as member variables

SystemC AMS Elaboration and TDF Simulation Cycle

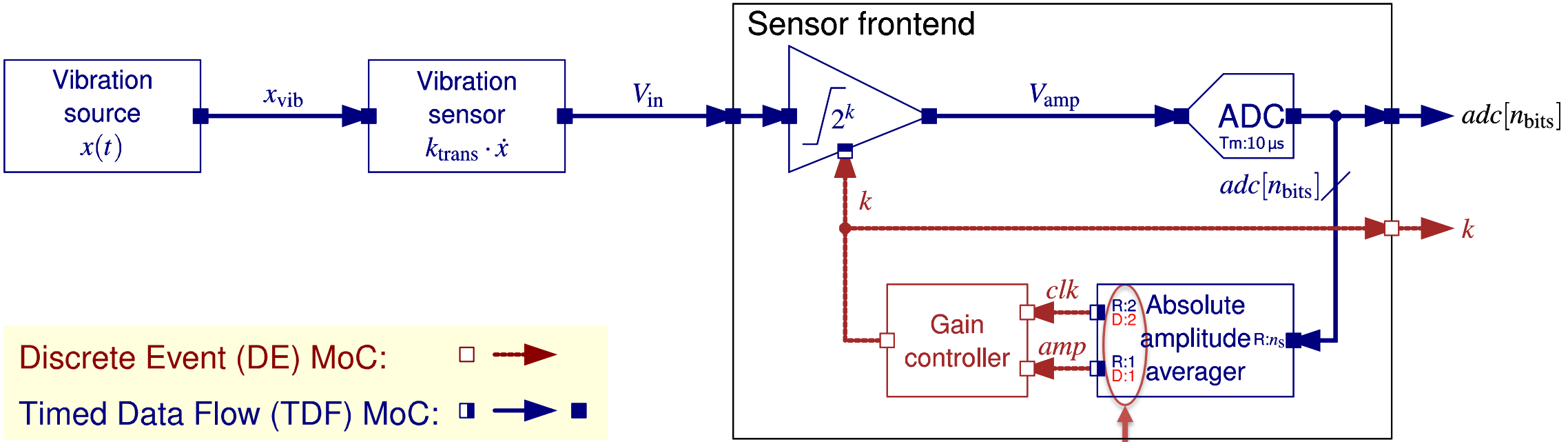


- Red callbacks only needed by TDF Modules supporting Dynamic TDF

Application Example: Vibration Sensor with Front End

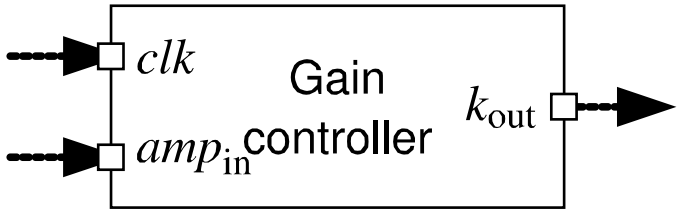


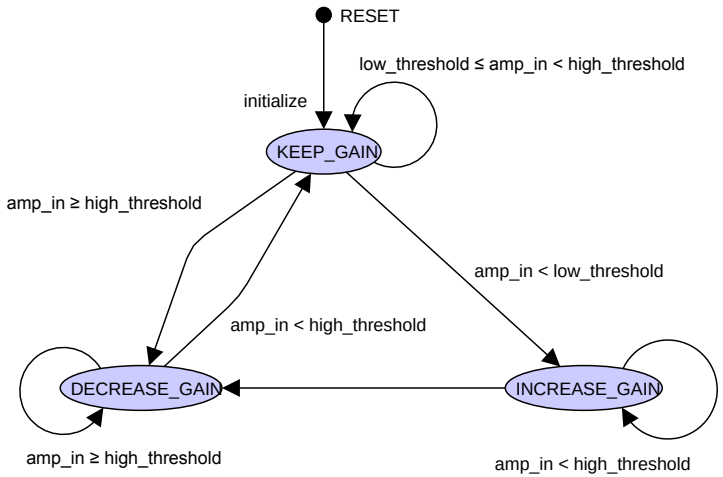
DE/TDF Model of the Vibration Sensor with Front End



- Do we need a delay somewhere? ⇝ Yes, to ensure correct TDF⟷DE synchronization!
- n_s samples consumed by averager per `processing()` (from t_m till $t_m + (n_s - 1) \cdot T_m$)
- DE writes need to happen afterwards:
  - 2 sample delay on clock
  - 1 sample delay on amplitude output

Gain Controller: DE Module






```
template<int NBits>
struct gain_controller : public sc_core::sc_module {
   sc_core::sc_in<bool> clk;
   sc_core::sc_in<sc_dt::sc_int<NBits> > amp_in;
   sc_core::sc_out<int> k_out;

   SC_HAS_PROCESS(gain_controller);

   explicit gain_controller(
             sc_core::sc_module_name nm,
             int low_threshold = 0.2 * ((1<<(NBits-1)) - 1), /* … */)
   : clk("clk"), …, low_threshold_(low_threshold), /* … */ {
     // Check consistency of model parameters using sc_assert()…
     // Register processes
     SC_METHOD(adapt_gain); sensitive << clk.pos();
   }
   void adapt_gain();
private:
   int k_;
   enum state_type {keep_gain, increase_gain,
                    decrease_gain};
   state_type state_;
   // …
};
```

Define input & output ports

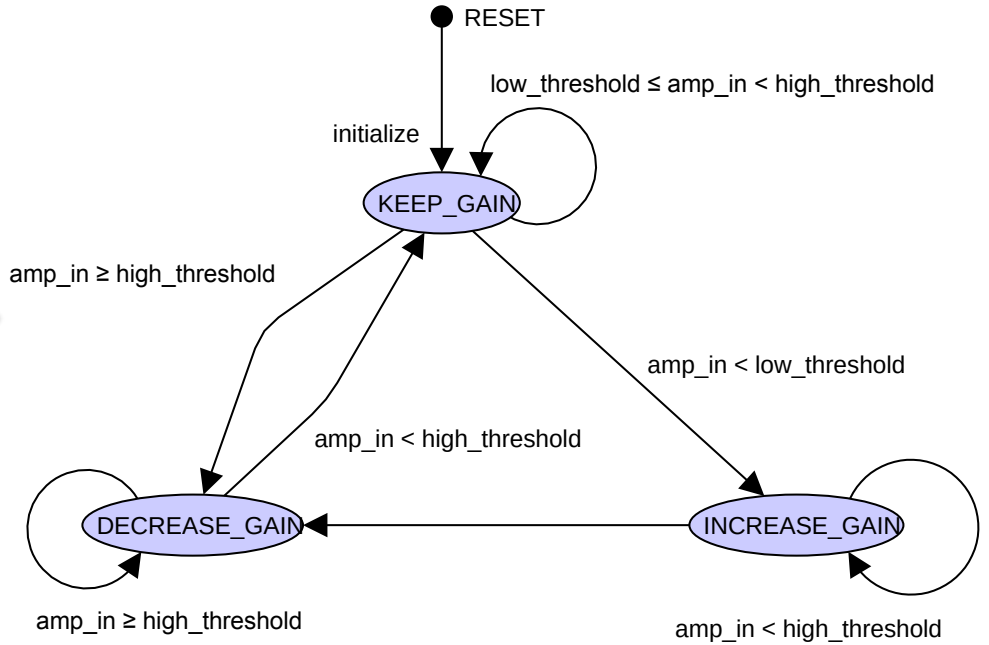
Construct and intitalize DE module

Register DE processes

Store internal state in private member variables

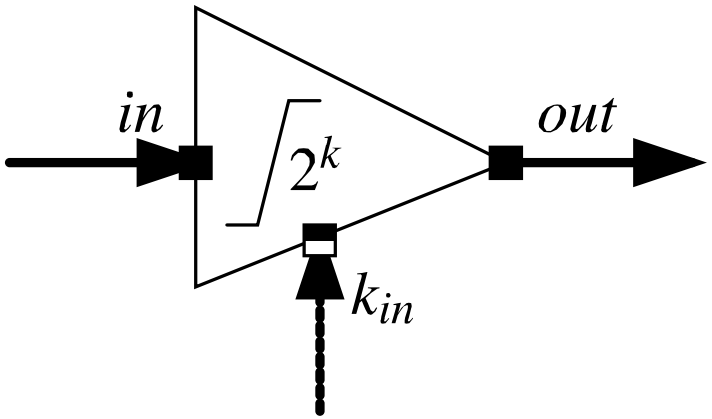
Gain Controller: Finite State Machine Behavior

```
template<int NBits> void gain_controller::adapt_gain() {
   switch (state_) {
   case keep_gain:
     if (amp_in.read() < low_threshold_) {
       state_ = increase_gain; ++k_; }
     else if (amp_in.read() >= high_threshold_) {
       state_ = decrease_gain; --k_; }
     break;
   case increase_gain: /* … */
   case decrease_gain: /* … */
   default:
     SC_REPORT_ERROR("/gain_controller", "Unexpected state");
   }

   if (k_ < k_min_) { k_ = k_min_; }
   if (k_ > k_max_) { k_ = k_max_; }
   k_out.write(k_);
}
```



Limit the minimum and maximum gain power k and write it to the output

Programmable Gain Amplifier: Single-Rate TDF Module

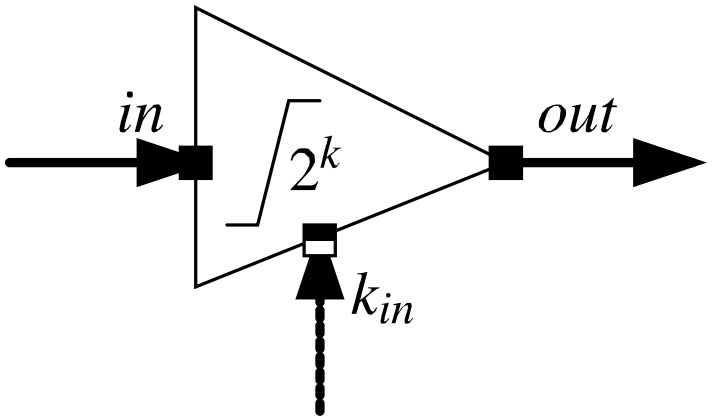


1. Construct & initialize TDF module

2. Set port rates and delays

```
struct programmable_gain_amplifier : public sca_tdf::sca_module {
   sca_tdf::sca_in<double> in;
   sca_tdf::sca_de::sca_in<int> k_in;
   sca_tdf::sca_out<double> out;

   programmable_gain_amplifier(
     sc_core::sc_module_name nm, double v_supply)
   : in("in"), k_in("k_in"), out("out"), v_supply_(v_supply) {
     sc_assert(v_supply > 0.0);
   }
   void set_attributes() {
     in.set_rate(1);
     k_in.set_rate(1);
     out.set_rate(1);
   }
   void initialize() {}
   void processing();
private:
   const double v_supply_; // Supply voltage limiting output.
};
```

Programmable Gain Amplifier: TDF Behavior

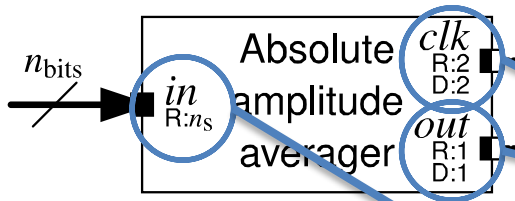


$$val := 2^{k_{in}} \cdot in$$

$$out := \begin{cases} V_{\text{supply}} & \text{für} \quad val > V_{\text{supply}} \\ -V_{\text{supply}} & \text{für} \quad val < -V_{\text{supply}} \\ val & \text{für} \quad |val| \leq V_{\text{supply}} \end{cases}$$

```
void
programmable_gain_amplifier::processing() {
    double k = k_in.read();
    // Amplify input value to output value.
    double val = std::pow(2.0, k) * in.read();
    // Test if output saturates.
    if (val > v_supply_) {
      out.write(v_supply_); }
    else if (val < -v_supply_) {
      out.write(-v_supply_); }
    else {
      out.write(val); }
}
```

Absolute Amplitude Averager: Multi-Rate TDF Module

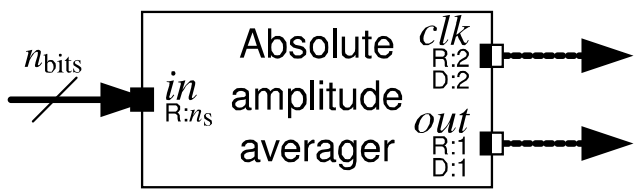



```
template<int NBits> struct abs_amplitude_averager : public sca_tdf::sca_module {
  sca_tdf::sca_in<sc_dt::sc_int<NBits> > in;
  sca_tdf::sca_de::sca_out<bool> clk;
  sca_tdf::sca_de::sca_out<sc_dt::sc_int<NBits> > out;

  explicit de_abs_amplitude_averager(sc_core::sc_module_name nm,
                                     long n_samples)
  : in("in"), clk("clk"), out("out"), n_samples_(n_samples) { /* … */ }
  void set_attributes() {
    in.set_rate(n_samples_); clk.set_rate(2);  out.set_rate(1);
    in.set_delay(0);         clk.set_delay(2); out.set_delay(1);
  }
  void initialize() {
    clk.initialize(true, 0); clk.initialize(false, 1);
    out.initialize(0);
  }
  void processing();
private:
  const long n_samples_;  // Number of averaged samples.
};
```

1. Set port rates and delays

2. Initialize delay samples

Absolute Amplitude Averager: Multi-Rate TDF Module

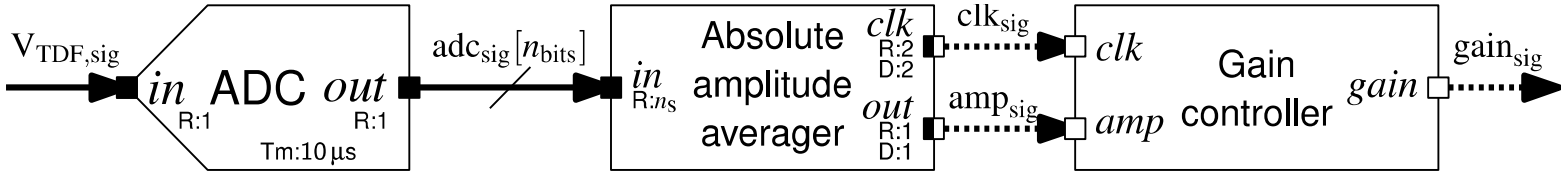


$$clk_{\{0,1\}} := \{t, f\}$$

$$out := \frac{\sum_0^{n_s-1} |in_i|}{n_s}$$

```
void abs_amplitude_averager::processing() {
  // Generate clock signal.
  clk.write(true, 0);
  clk.write(false, 1);
  // Calculate and output average of
  // absolute amplitudes.
  long sum = 0;
  for (long i = 0; i < n_samples_; ++i) {
    sum += std::labs(in.read(i));
  }
  long avg = sum / n_samples_;
  out.write(avg);
}
```

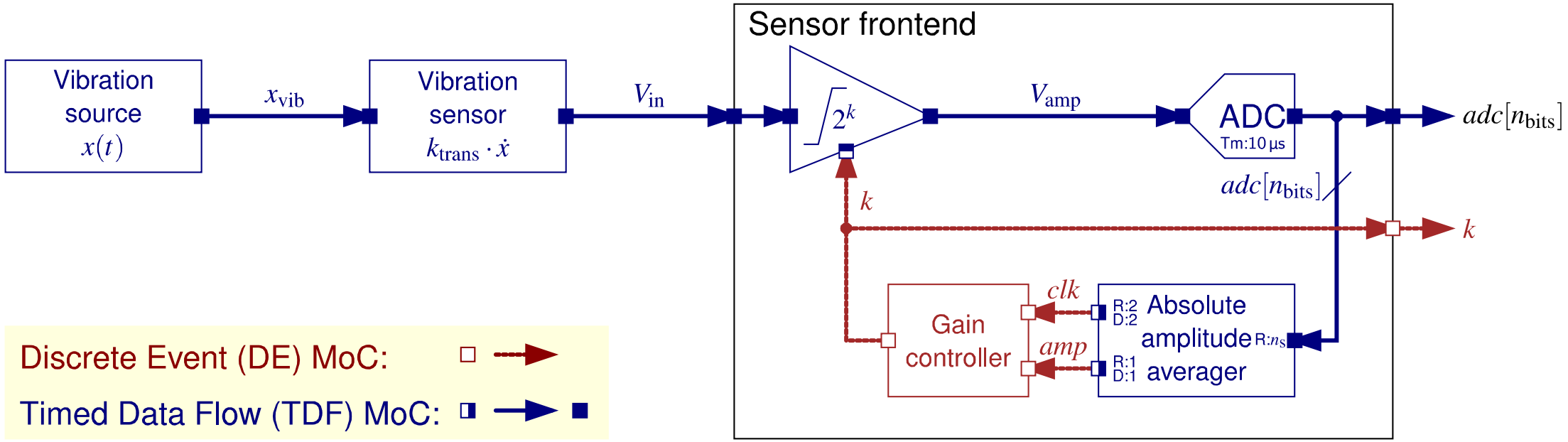
Vibration Sensor System: Structural Composition



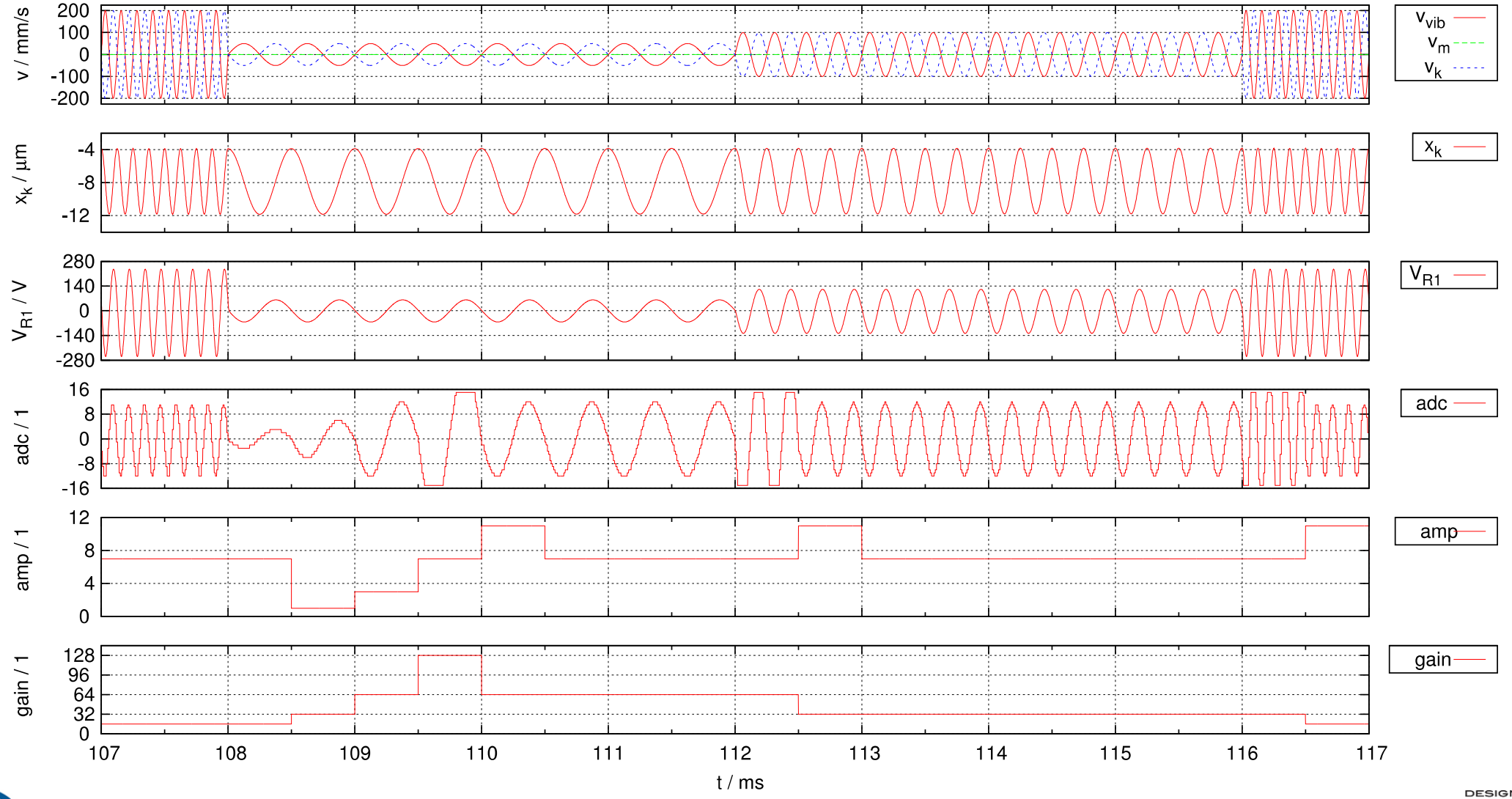

```
// Define signals for module interconnection.
sca_tdf::sca_signal<double>                  v_tdf_sig("v_tdf_sig");
sca_tdf::sca_signal<sc_dt::sc_int<NBitsADC> > adc_sig("adc_sig");
sc_core::sc_signal<bool>                     clk_sig("clk_sig");
sca_tdf::sca_signal<sc_dt::sc_int<NBitsADC> > amp_sig("amp_sig");
sc_core::sc_signal<int>                      k_sig(”k_sig");
// …
// Instantiate the modules and bind their ports to the signals.
ad_converter<NBitsADC> adc_1("adc_1", V_pp);
  adc_1.set_timestep(t_adc);
  adc_1.in(v_tdf_sig); adc_1.out(adc_sig);

abs_amplitude_averager<NBitsADC> avg_1("avg_1", avg_n_samples);
  avg_1.in(adc_sig); avg_1.clk(clk_sig); avg_1.out(amp_sig);

gain_controller<NBitsADC>
  gain_ctrl_1("gain_ctrl_1", amp_low_threshold, amp_high_threshold,
              k_0, k_min, k_max);
  gain_ctrl_1.clk(clk_sig); gain_ctrl_1.amp(amp_sig); gain_ctrl_1.k_out(k_sig);
// …
```

1. Signal instantiation
2. Module instantiation
3. Port to signal binding
4. Assign module time step (if not yet done)

DE/TDF Model of the Vibration Sensor with Front End



Simulation Results



Embedding Continuous-Time Behavior into TDF Modules

TDF modules may include (switched) linear CT behavior:

- **sca_ltf_nd**: Laplace Transfer Functions (LTF) in numerator-denominator form:

$$H(s) = k \cdot \frac{\sum_{i=0}^{M-1} num_i \cdot s^i}{\sum_{k=0}^{N-1} den_k \cdot s^k} \cdot e^{-s \cdot delay}$$

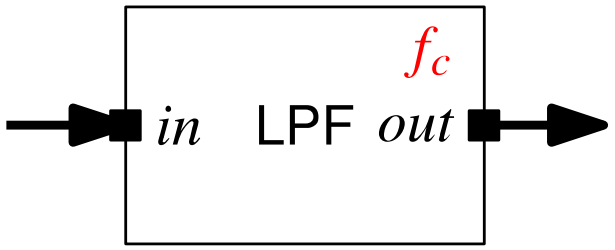
- **sca_ltf_zp**: Laplace Transfer Functions (LTF) in zero pole form:

$$H(s) = k \cdot \frac{\prod_{i=0}^{M-1}(s - zeros_k)}{\prod_{k=0}^{N-1}(s - poles_k)} \cdot e^{-s \cdot delay}$$

- **sca_ss**: state space equation system:

$$\begin{aligned} \frac{ds(t)}{dt} &= \mathbf{A} \cdot s(t) + B \cdot x(t - delay) \\ y(t) &= \mathbf{C} \cdot s(t) + \mathbf{D} \cdot x(t - delay) \end{aligned}$$

TDF CT Behavior Example: Low-Pass Filter



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

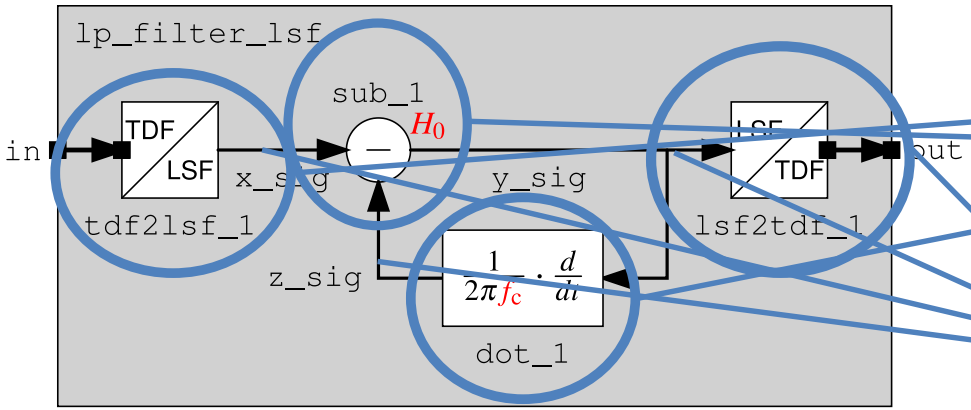
```
struct lp_filter_tdf : sca_tdf::sca_module
{
   sca_tdf::sca_in<double>  in;
   sca_tdf::sca_out<double> out;

   sca_tdf::sca_ltf_nd ltf;
   sca_util::sca_vector<double> num, den;

   lp_filter_tdf(const sc_core::sc_module_name&,
                 double fc) {
     num(0) = 1.0;
     den(0) = 1.0; den(1) = 1.0/(2.0*M_PI*fc);
   }

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

}; // struct lp_filter_tdf
```

1. Instantiate LTF solver
2. Vectors to hold LTF coeffs
3. Initialize LTF coeffs
4. Call LTF solver for next sample of CT behavior

SystemC AMS MoCs for Continuous-Time Modeling

Linear Signal Flow (LSF)

- Primitive-based
- Block diagrams
- Non-conservative linear behavior

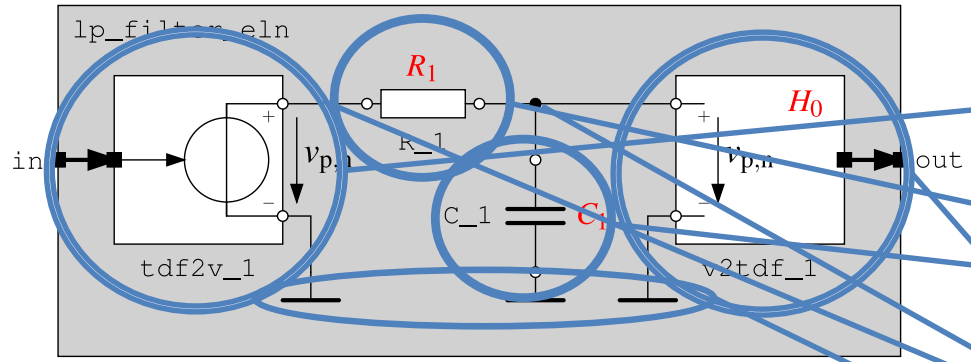


Used LSF primitives:

- tdf2lsf_1: sca_lsf::sca_tdf::sca_source
- sub_1: sca_lsf::sca_sub
- dot_1: sca_lsf::sca_dot
- lsf2tdf_1: sca_lsf::sca_tdf::sca_sink
- sca_lsf::sca_signal

Electrical Linear Networks (ELN)

- Primitive-based
- Electrical circuits
- Conservative linear behavior



Used ELN primitives:

- tdf2v_1: sca_eln::sca_tdf::sca_vsource
- R_1: sca_eln::sca_r
- C_1: sca_eln::sca_c
- v2tdf_1: sca_eln::sca_tdf::vsink
- sca_eln::sca_node
- sca_eln::sca_node_ref

Outlook on Dynamic TDF

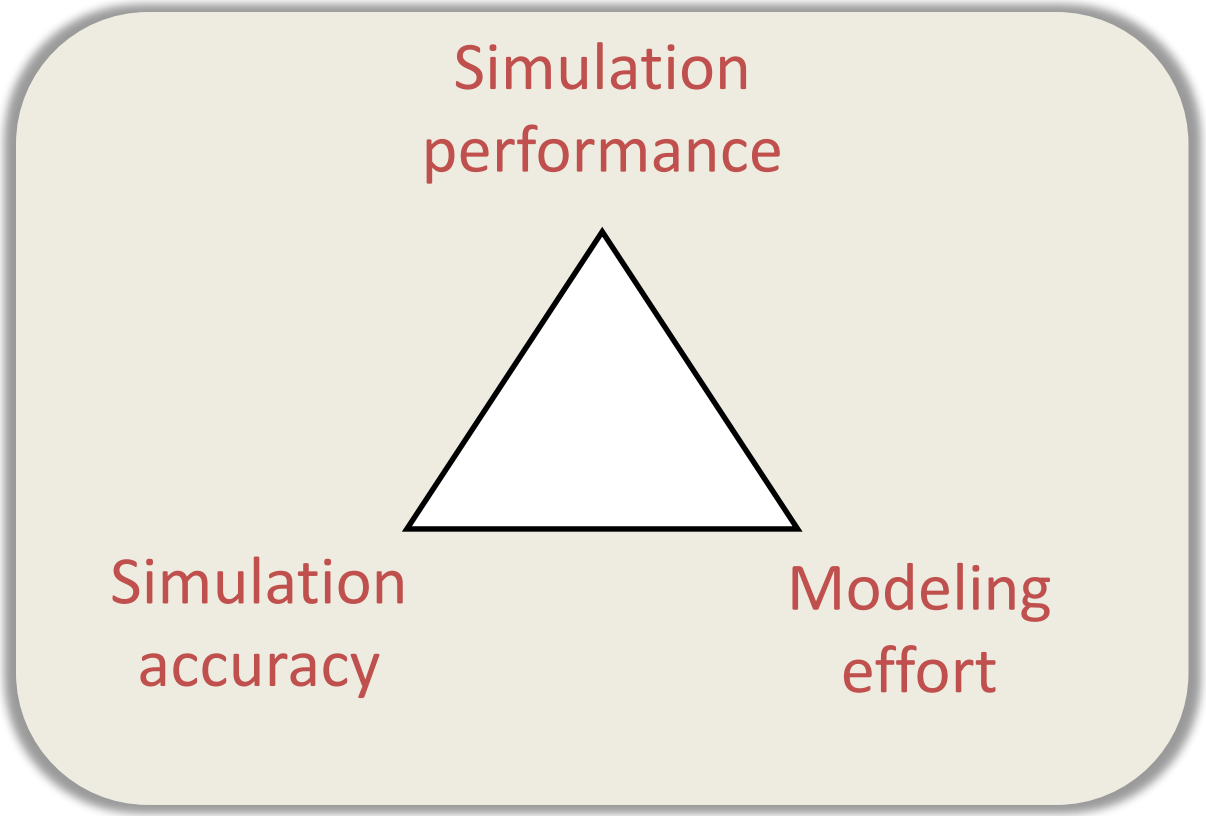
- TDF attributes in the shown TDF models are static for whole simulation
- Dynamic TDF features were added in SystemC AMS 2.0 to facilitate:
  - Abstract modelling of sporadically changing signals
    - E.g. power management that switches on/off AMS subsystems
  - Abstract description of reactive behaviour
    - AMS computations driven by events or transactions
  - Capture behaviour where frequencies (and time steps) change dynamically
    - Often the case for clock recovery circuits or capturing jitter
  - Modelling systems with varying (data) rates
    - E.g. multi-standard / software-defined radios
- To this end:
  - TDF modules may be marked to accept_attribute_changes() and does_attribute_changes()
  - TDF attributes may be modified in change_attributes() callback during simulation
  - Next processing() activation may be modified using request_next_activation() after a specified event or time out.

Resources

- IEEE Std 1666.1-2016: https://standards.ieee.org/findstds/standard/1666.1-2016.html
- Accellera SystemC AMSWG: http://www.accellera.org/activities/working-groups/systemc-ams
- SystemC-AMS homepage: http://www.systemc-ams.org/
- SystemC-AMS 2.1 PoC implementation from COSEDA Technologies: http://www.coseda-tech.com/systemc-ams-proof-of-concept
- Accellera Systems Initiative SystemC Community: http://www.accellera.org/community/systemc/
- Accellera Systems Initiative Forums: http://forums.accellera.org/

SYSTEMC AMS MODELING TECHNIQUES

SystemC AMS Modeling Techniques – Modeling and Simulation Tradeoff



SystemC AMS Modeling Techniques
Modeling Questions

- What are the model **use cases** ?

- What are the **relevant effects** ?
  - Restrictions will result in faster models and lower modeling effort
  - To general model requirements will make the model expensive, error prone and slow – and thus may useless

SystemC AMS Modeling Techniques Performance

- Usually mainly influenced by the **number of activations**
- SystemC **thread** activation is more expensive than **method** activation which is more expensive than a SystemC-AMS **dataflow** module activation
- Effort for solving linear DAE systems increases approx. **linear with** the number of equations – for ELN nearly with the **number of nodes**
- **Changing the equation system** (e.g. switch, change a resistor, … or the timestep) is **expensive** (approx. 10 times slower than a normal timestep)
- Reduce effort in **often activated modules**

SystemC AMS Modeling Techniques
Example: Leakage: Integrator Imperfections

- Ideal Integrator

$$H(z) = \frac{\beta}{1 - z^{-1}}$$

$$H(z) = \frac{\beta z^{-1}}{1 - z^{-1}}$$

$$\beta = C_S / C_F$$

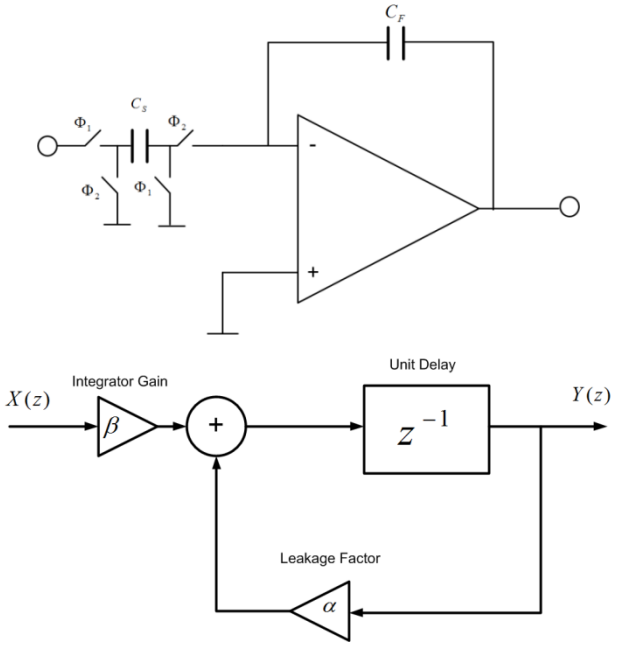
- Leaky Integrator

$$H(z) = \frac{\beta}{1 - \alpha z^{-1}}$$

$$H(z) = \frac{\beta z^{-1}}{1 - \alpha z^{-1}}$$

$$\alpha = (1 - \mu)$$

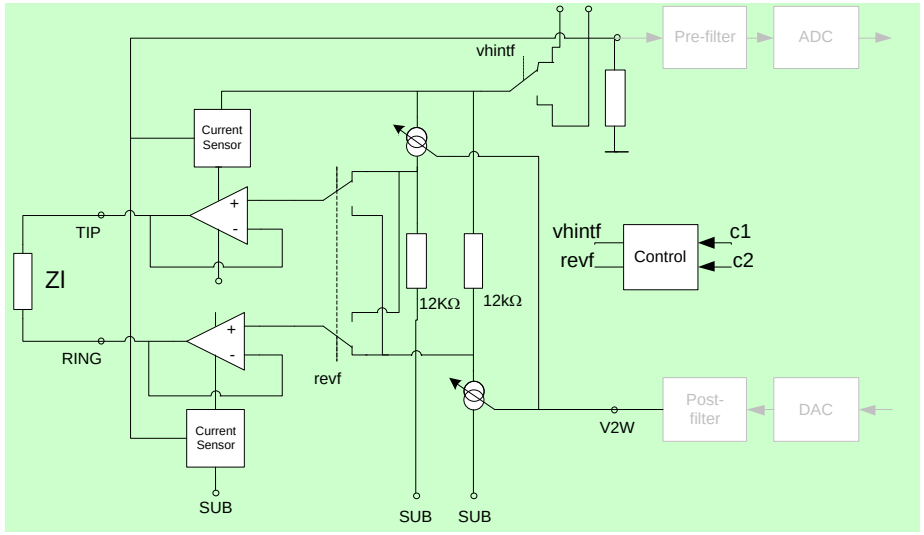
$$\mu = \frac{A_{CL}}{A_{OL}} = \frac{C_S / C_F}{A_{OL}}$$

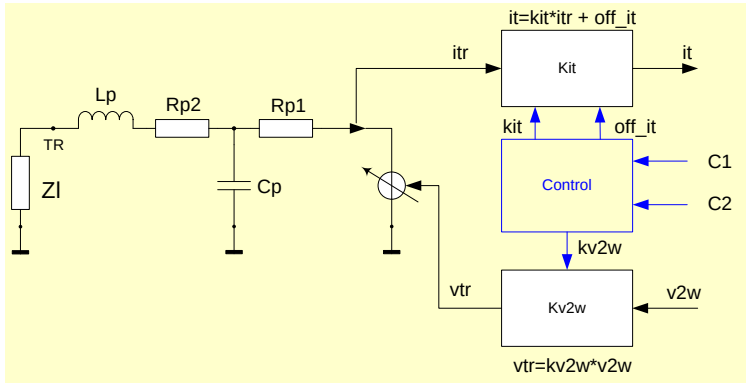



```
void integrator::processing()
{
    y.write(y_last);
    y_last=alpha*y_last + beta*x.read();
}
```

Source: Sumit Adhikari TU Vienna

SystemC AMS Modeling Techniques Abstraction

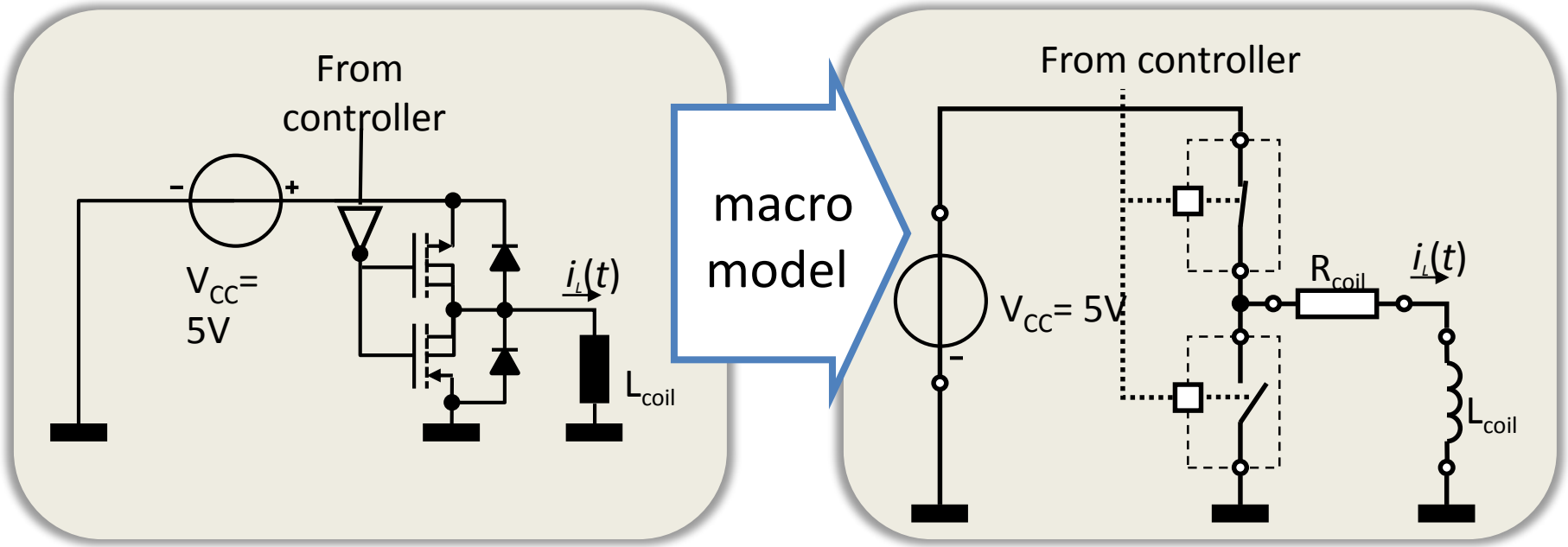




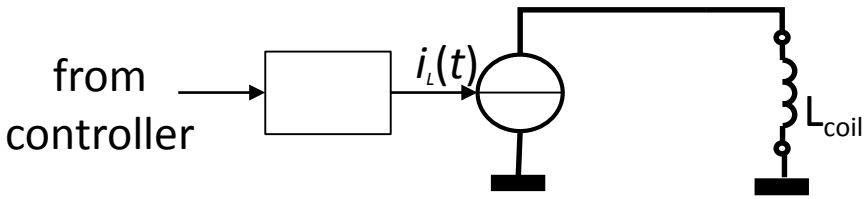
ELN / TDF Model

```
SCA_TDF_MODULE(kv2w)
{
    sca_tdf::sca_in<double>     v2w;
    sca_tdf::sca_out<double> vtr;

    // control / DE inport
    sca_tdf::sc_in<double> k_v2w;

    void processing();

    SCA_CTOR(kv2w)
    {
    }
};
void kv2w::processing()
{
    double v2w_tmp = v2w.read();
    double vtr_tmp;

    vtr_tmp = k_v2w.read() * v2w_tmp;

    vtr.write(vtr_tmp);
}
```

SystemC AMS Modeling Techniques Abstraction PWM

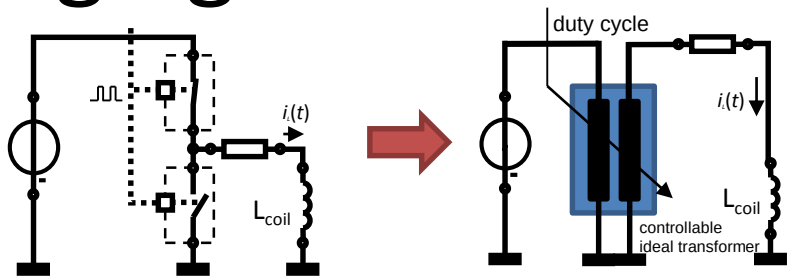


Source: Christoph Grimm TU Vienna



SystemC AMS Modeling Techniques
PWM Averaging

- Numerous state of the art power devices based on pulse width modulation
- Principle: by switching with an high frequency a voltage or current is controlled
- High switching frequency -> long simulation time
- Idea: Instead of modelling each switching event, the average is modelled
- A performance increase by a factor ~100 can be achieved

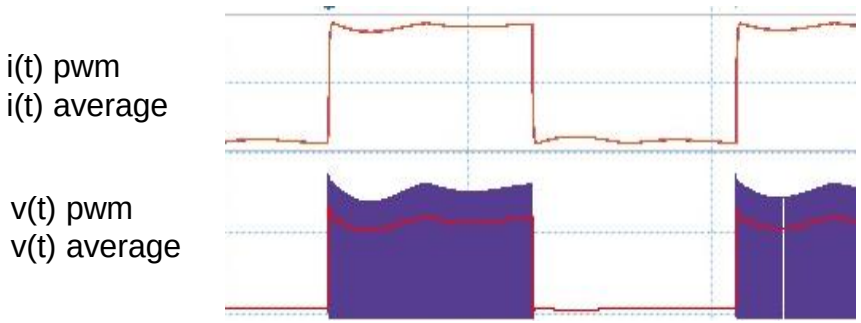


$$I_L = \frac{1}{L} \cdot \int V_L dt$$

$$V_{in} \cdot I_{in} = V_{LR} \cdot I_L$$

$$V_{in} \cdot I_{in} = duty \cdot V_{in} \cdot I_L$$

$$I_{in} = duty \cdot I_L$$
$$V_{LR} = duty \cdot V_{in}$$



SystemC AMS AC Modeling

- AC-analysis:
  - Calculates linear complex equation system stimulated by AC-sources
  - -> Linear frequency dependent transfer function (bode diagram)
- 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 callback method ***ac_processing*** (otherwise the out values are assumed zero)
  - This transfer function can depend on the current time domain state (e.g. the setting of a control signal)

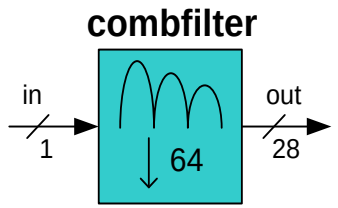
SystemC AMS AC Modeling

Frequency Domain Description for TDF Models

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

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

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

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

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

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

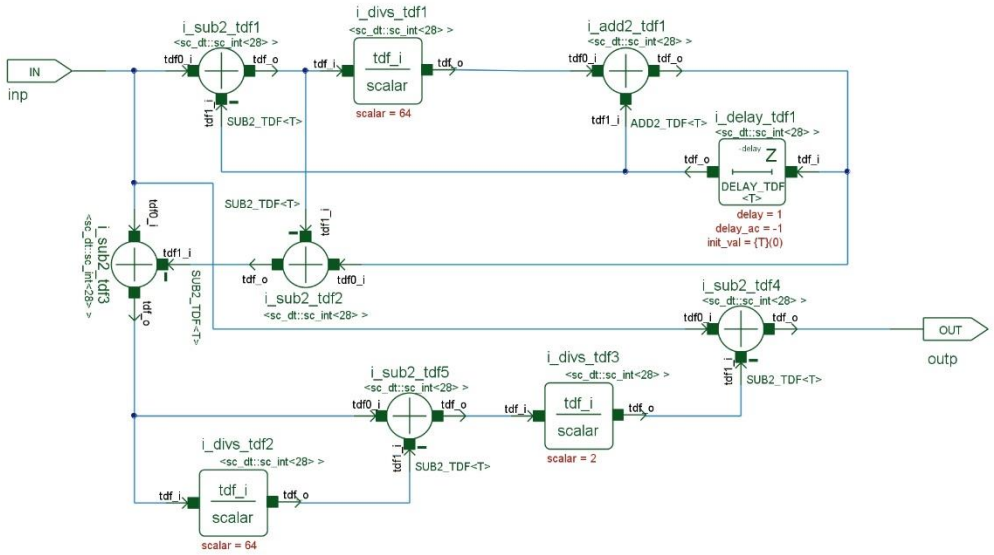
$$H(z) = \left(\frac{1 - z^{-k}}{1 - z^{-1}} \right)^{n}$$



$$z = e^{j2\pi f/f_s}$$

SystemC AMS AC Modeling

Example: AC – Modelling TDF




```
void add2::ac_processing()
{
  sca_ac(outp) = sca_ac(inp1) + sca_ac(inp2);
}

void sub2::ac_processing()
{
  sca_ac(outp) = sca_ac(inp1) - sca_ac(inp2);
}
```

- If all modules in the data path have a AC description, the hierarchical model will have implicitly an AC description

```
void delay::ac_processing()
{
    sca_ac(outp) = sca_ac_z(sample_delay) * sca_ac(inp);
}

void divs::ac_processing()
{
    sca_ac(outp) = sca_ac(inp) / scalar;
}
```

SystemC AMS AC Modeling

Example: Noise Source

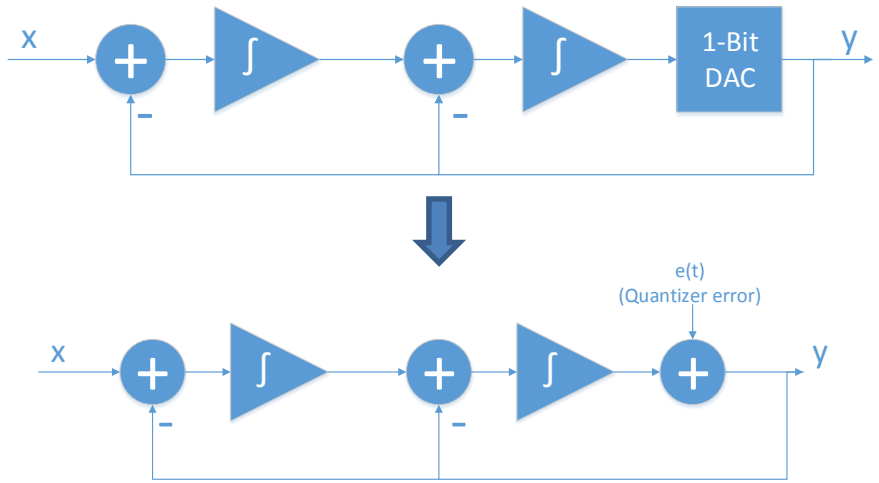
```
double add_noise::noise_level(double bw)
{
  // P=V*I; z=V/I -> P=V**2/z
  // dbmHz=10*log(P/1mW)
  // P/Hz=10**(dbm/10)*1mw;
  // V = sqrt(10**(dbm/10)*bw*z*1mW)
  return
     sqrt(pow(10.0,p.dbmHz/10.0)*p.z*1e-3*bw);
}
```

```
void add_noise::processing()
{
  //(measurement) bandwith for time domain is
  //nyquist frequency of the timestep
  double  bw;
  bw=1.0/(2.0*get_timestep().to_seconds());
  double level=noise_level(bw);

  outp=get_gausian_random_number
                    (0.0,level,s.seed) +
                    inp.read();
}
```

```
void add_noise::ac_processing()
{
  sca_ac(outp)=sca_ac(inp);
  sca_ac_noise(outp)=noise_level(1.0);
}
```

SystemC AMS AC Modeling

Example: SDADC – AC Modelling



- $Y(z)=X(z)z^{-1} + E(z)(1-z^{-1})^2$
- $E(f) = \frac{q}{\sqrt{12}} \cdot \sqrt{\frac{2}{f_s}}$
- q=1.0 for a 1Bit DAC

```
void sdadc::ac_processing()
{

//noise transfer function
sca_complex ntf=pow(1.0-sca_ac_z(-1), 2);

  //quantizer noise
  //random uncorrelated noise in 1/sqrt(Hz)
  sca_complex e=
     1.0/sqrt(12.0) *

sqrt(2.0*get_timestep().to_seconds());


  sca_ac(outp) = sca_ac_z(-1)*sca_ac(inp);

  //e is effective value – we use the
  //peak value -> sqrt(2)
  sca_ac_noise(outp)=ntf*e*sqrt(2.0);
}
```

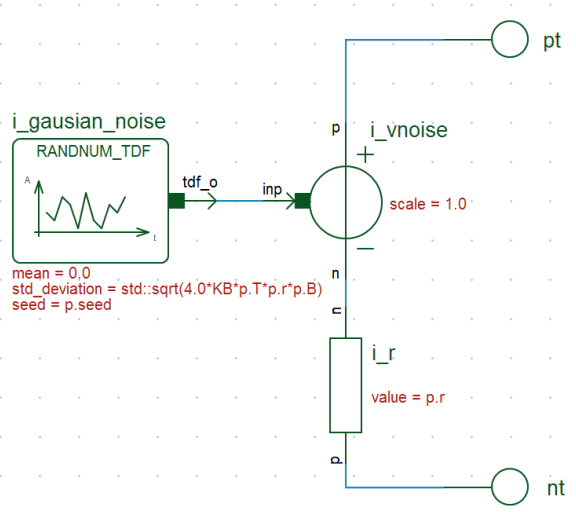
SystemC AMS AC Modeling

Example: Thermal Noise for ELN

- Thermal noise is a white (equally distributed power over the frequency) Gaussian (normal distributed values) Noise
- Results from the thermal movement of carriers
- Also called Nyquist or Johnson noise
- The effective noise voltage across a resistor is:

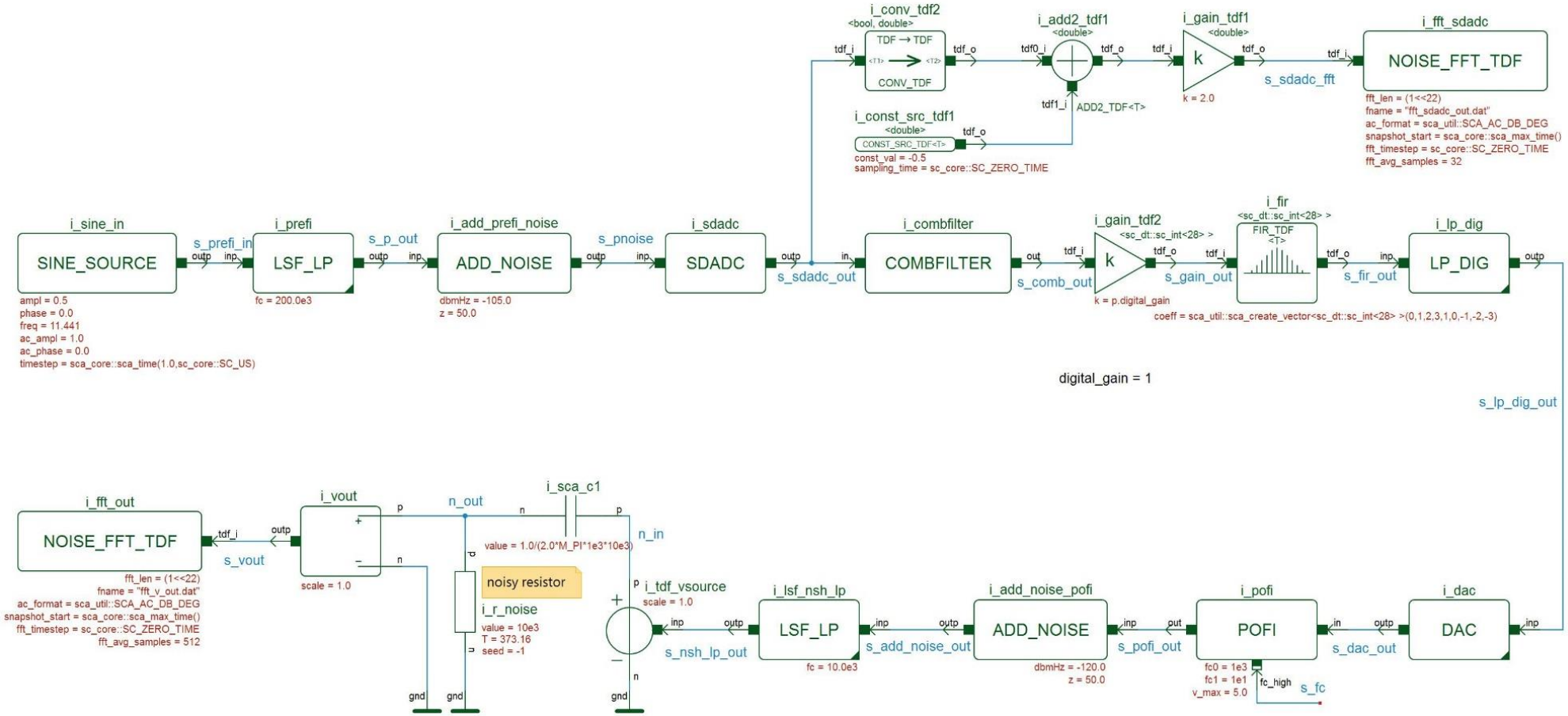
$$V_{noise} = \sqrt{4 \cdot k_B \cdot T \cdot R \cdot B}$$

  - kB – Boltzmann constant, R – resistance, B – Measurement bandwidth



SystemC AMS AC Modeling

Example



SystemC AMS AC Modeling

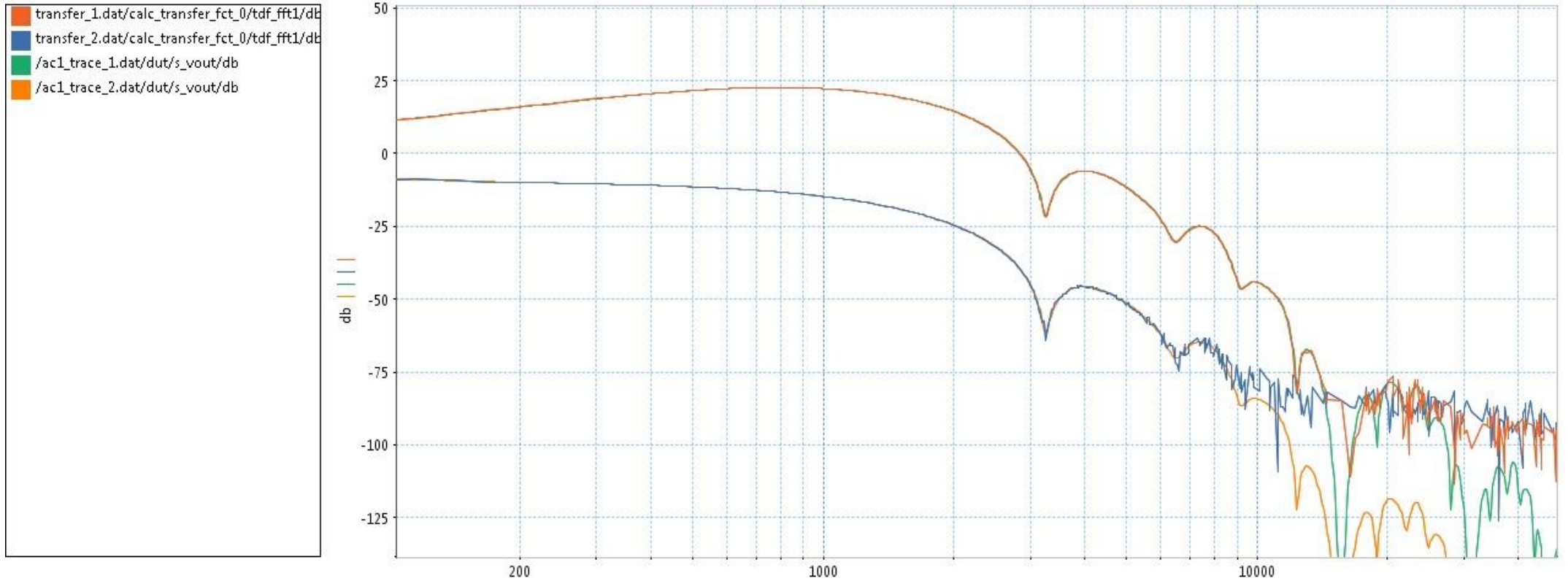
Example: Testsequence

```
…
tf=sca_create_tabular_trace_file("ac_trace_1.dat");

 //store ac and noise result as dB and degree
 tf->set_mode(sca_ac_format(sca_util::SCA_AC_DB_DEG));

 //store the noise besides the sum for
 //each noise source separately
 tf->set_mode(sca_noise_format(sca_util::SCA_NOISE_ALL));

 //trace signals
 sca_trace(tf,sig1,“sig1"); …
…

//start AC with initial control signal setting
sca_ac_start(100.0, 500.0e3, 1000, SCA_LOG);

//set control signal to true and
//start time domain simulation to
//propagate the control signal
dut->s_fc.write(true);
```

```
//write results to new time domain result file
tf->reopen(“time_domain.dat");

sc_start(1.0,SC_MS);

//open new AC result file
tf->reopen("ac_trace_2.dat");

sca_ac_start(100.0, 500.0e3, 1000, SCA_LOG);

dut->s_fc.write(false);

//reopen time domain result file (append)
tf->reopen(“time_domain.dat",std::ios::app);

sc_start(1.0,SC_MS);

//open new ac result file
tf->reopen("ac_noise_trace_1.dat");

//start ac noise simulation
sca_ac_noise_start(10.0,500.0e3,1000,SCA_LOG);
```

SystemC AMS AC Modeling

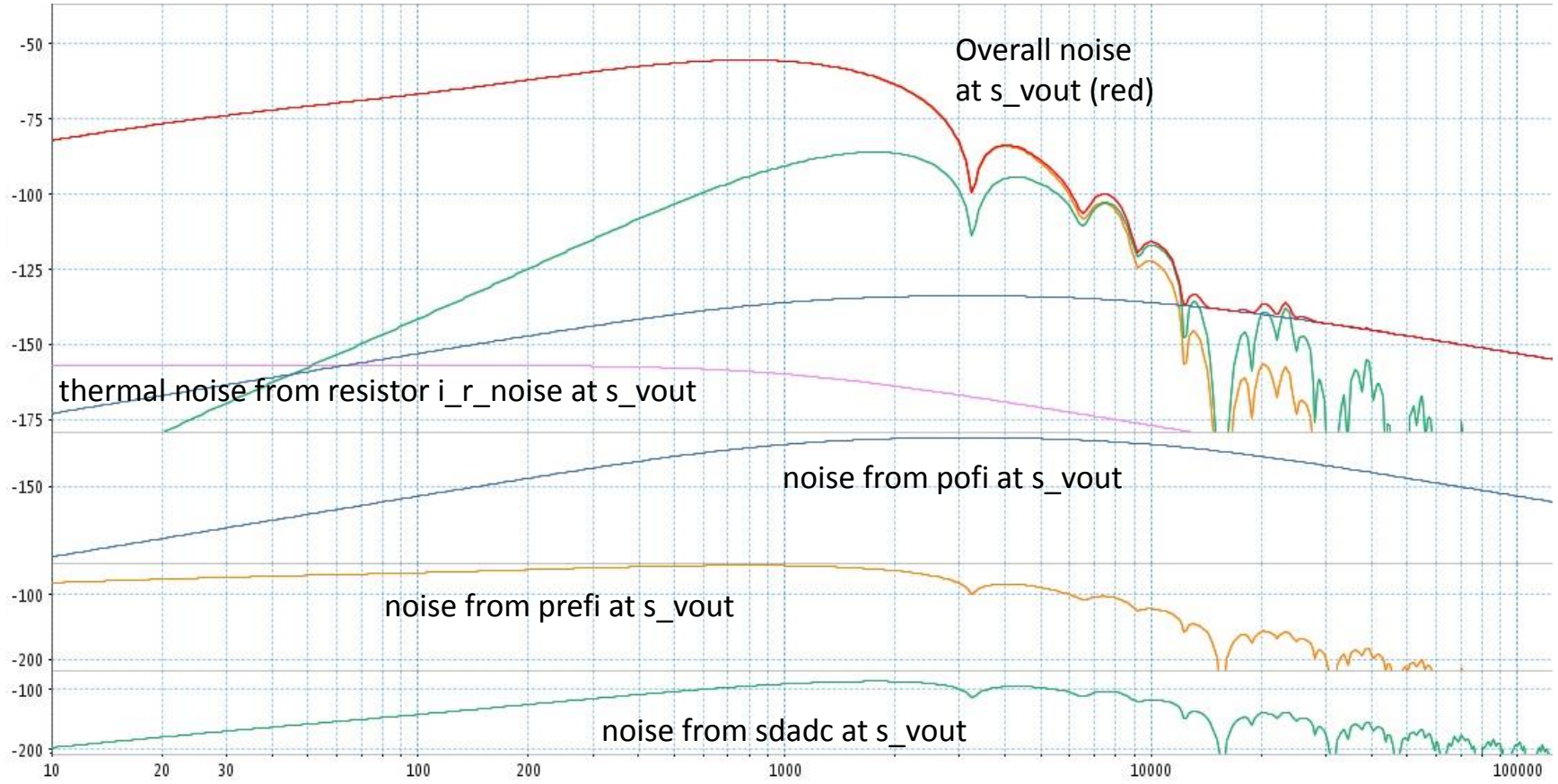
Result Comparison AC and Time domain simulation



SystemC AMS AC Modeling

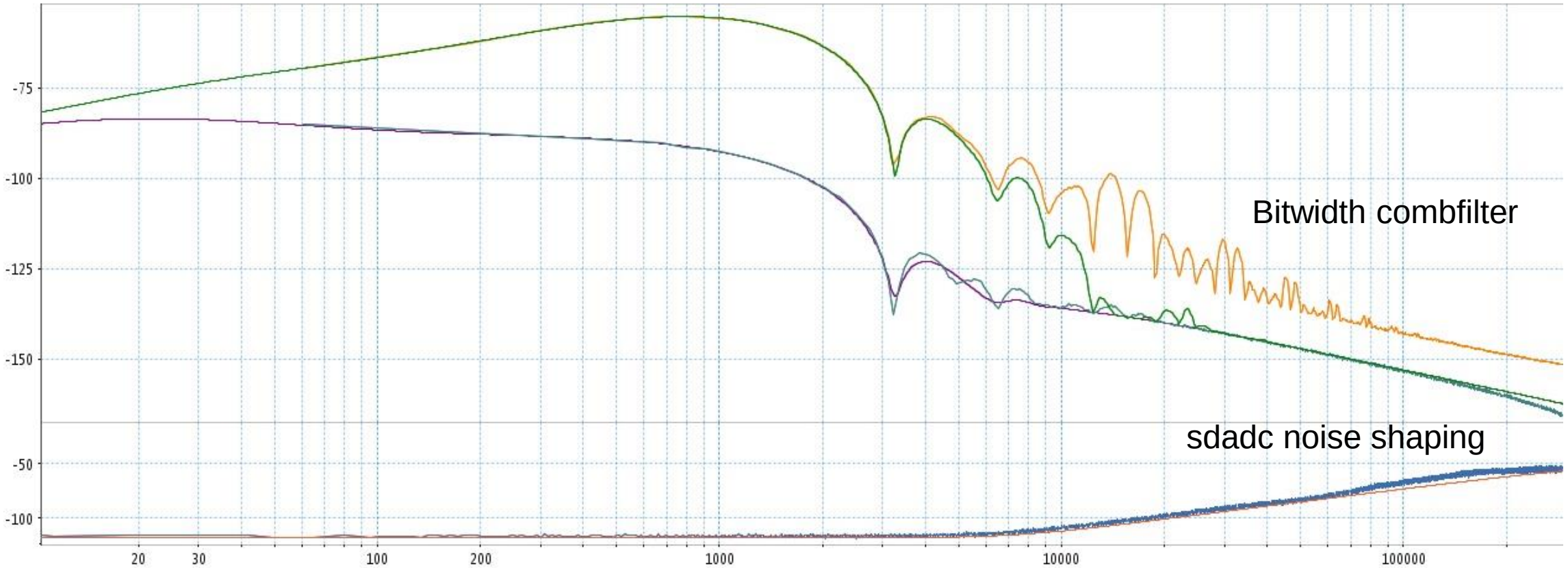
Analysis of AC noise simulation results

```
tf->set_mode(sca_noise_format(sca_util::SCA_NOISE_ALL) );
```



SystemC AMS AC Modeling

Comparison AC noise and time domain simulation

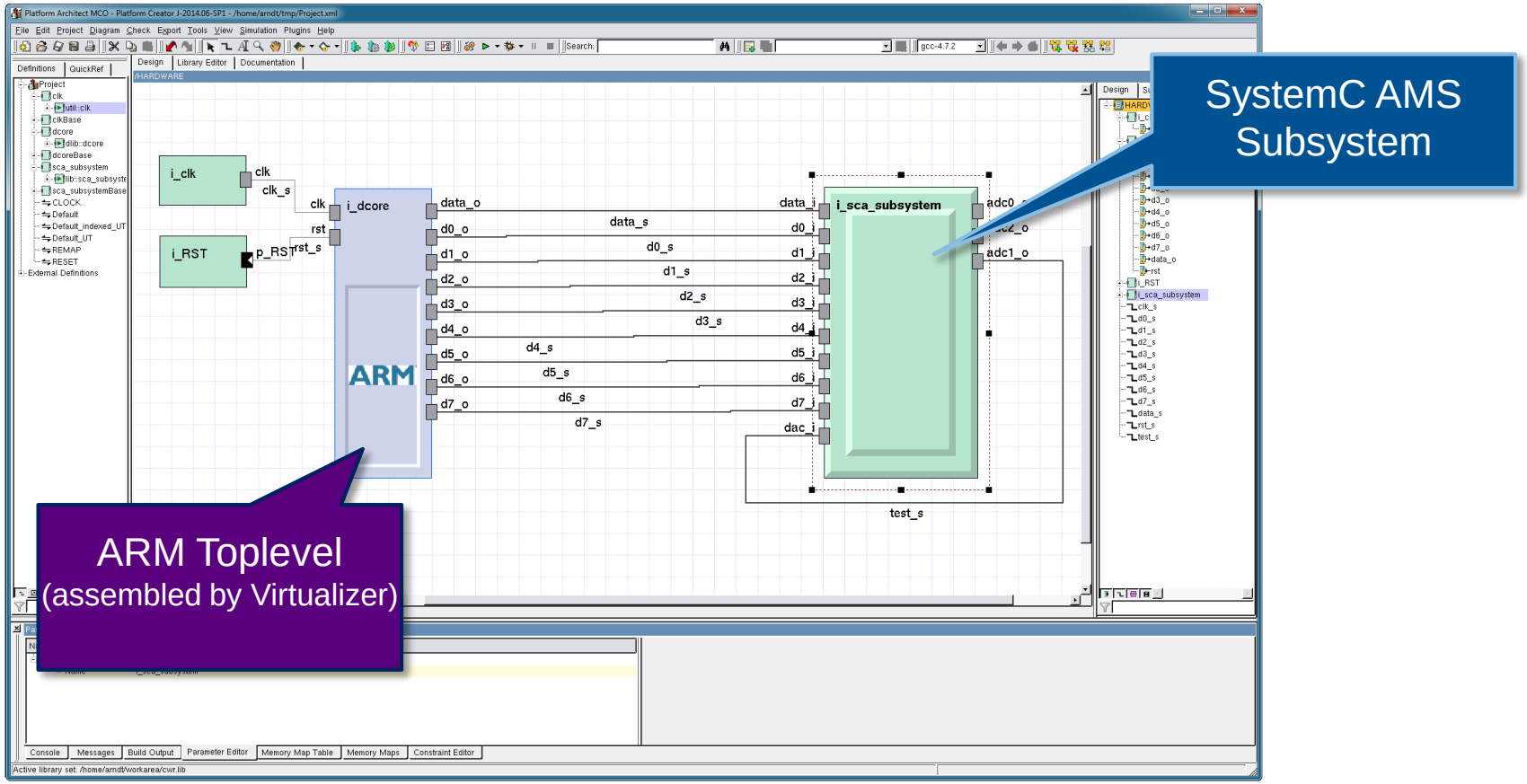


DESIGN FLOW INTEGRATION AND TOOLS

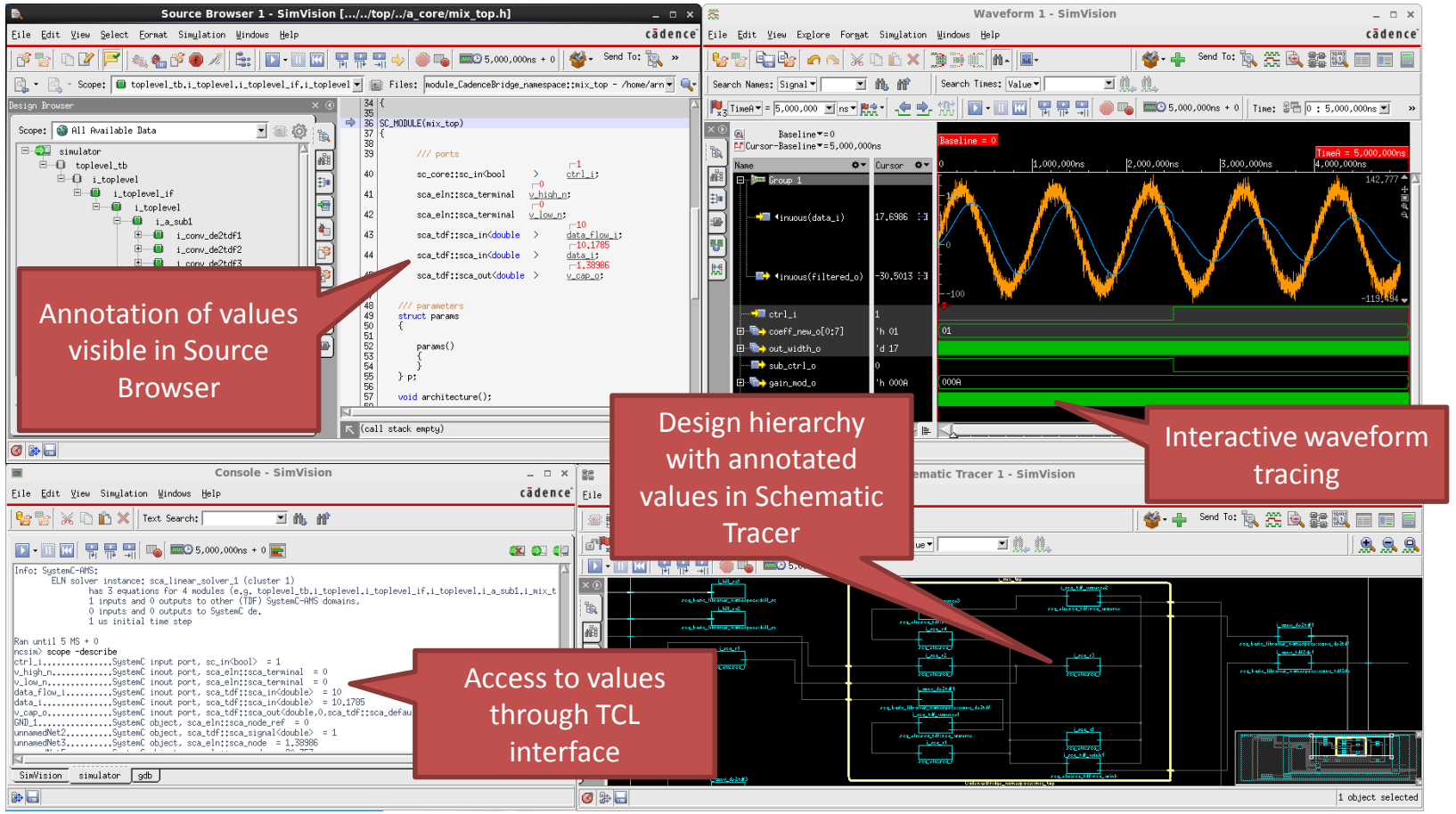
SystemC AMS Designflow Integration and Tools

- SystemC AMS PoC available under Apache license
  - http://www.coseda-tech.com/systemc-ams-proof-of-concept
- SystemC AMS must be complied on top of SystemC
- SystemC AMS can be compiled with commercial SystemC simulators

SystemC AMS in Synopsys Virtualizer



SystemC AMS in Cadence



SystemC AMS Model Re-use

SystemC AMS Model

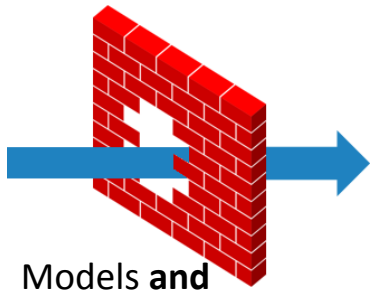
(System)Verilog-(AMS) Model

Overall system and application scenarios, functional verification

Algorithm

Concept-/ System Level

Architecture



Models **and** Verification scenarios including fault injection

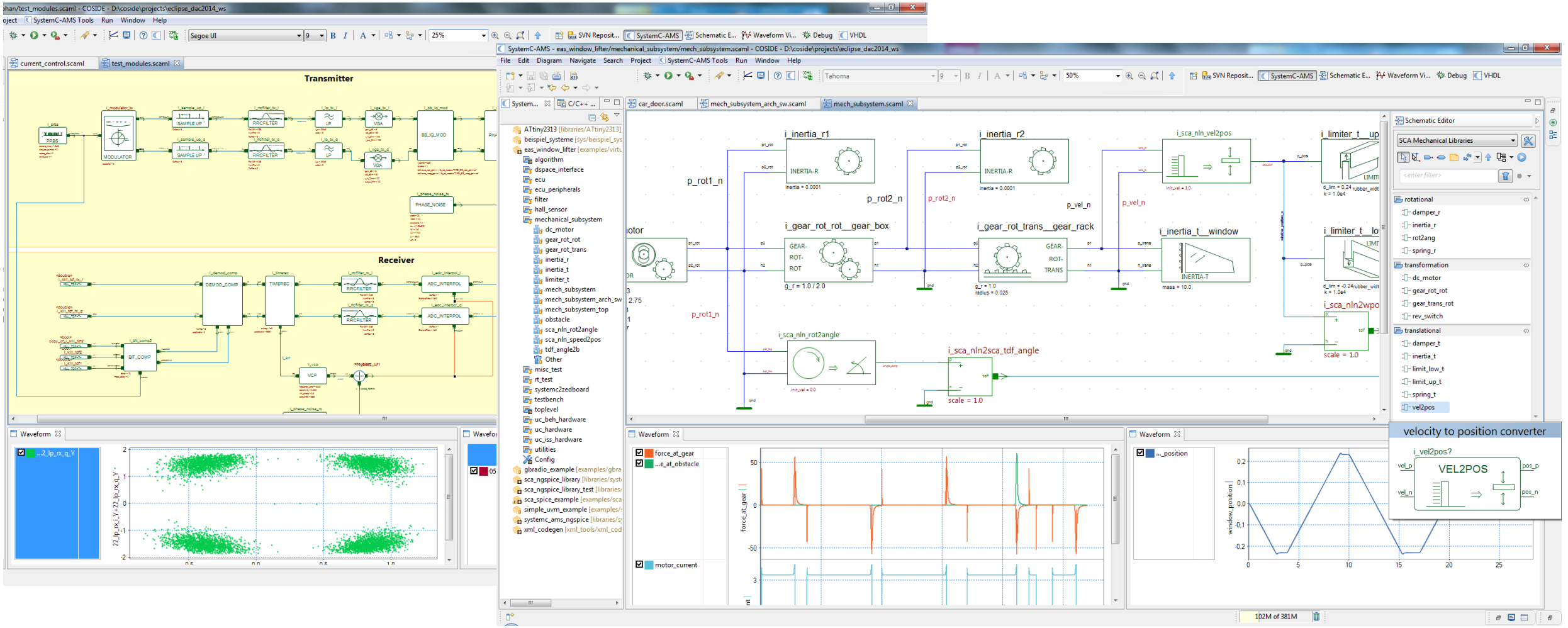
Architecture

Implementation (RTL/Transistor)

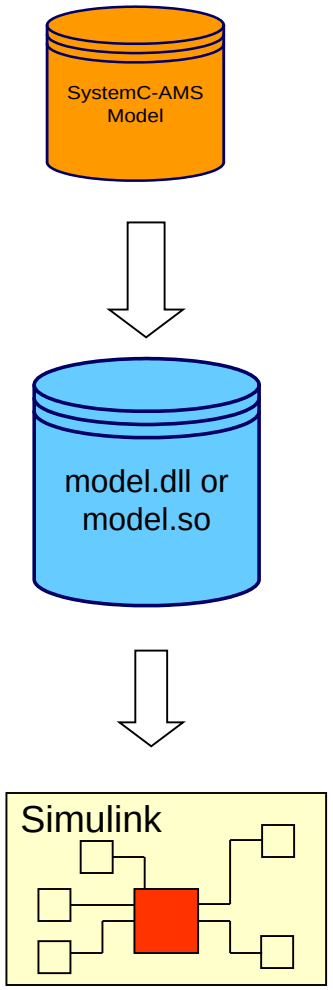
Silicon

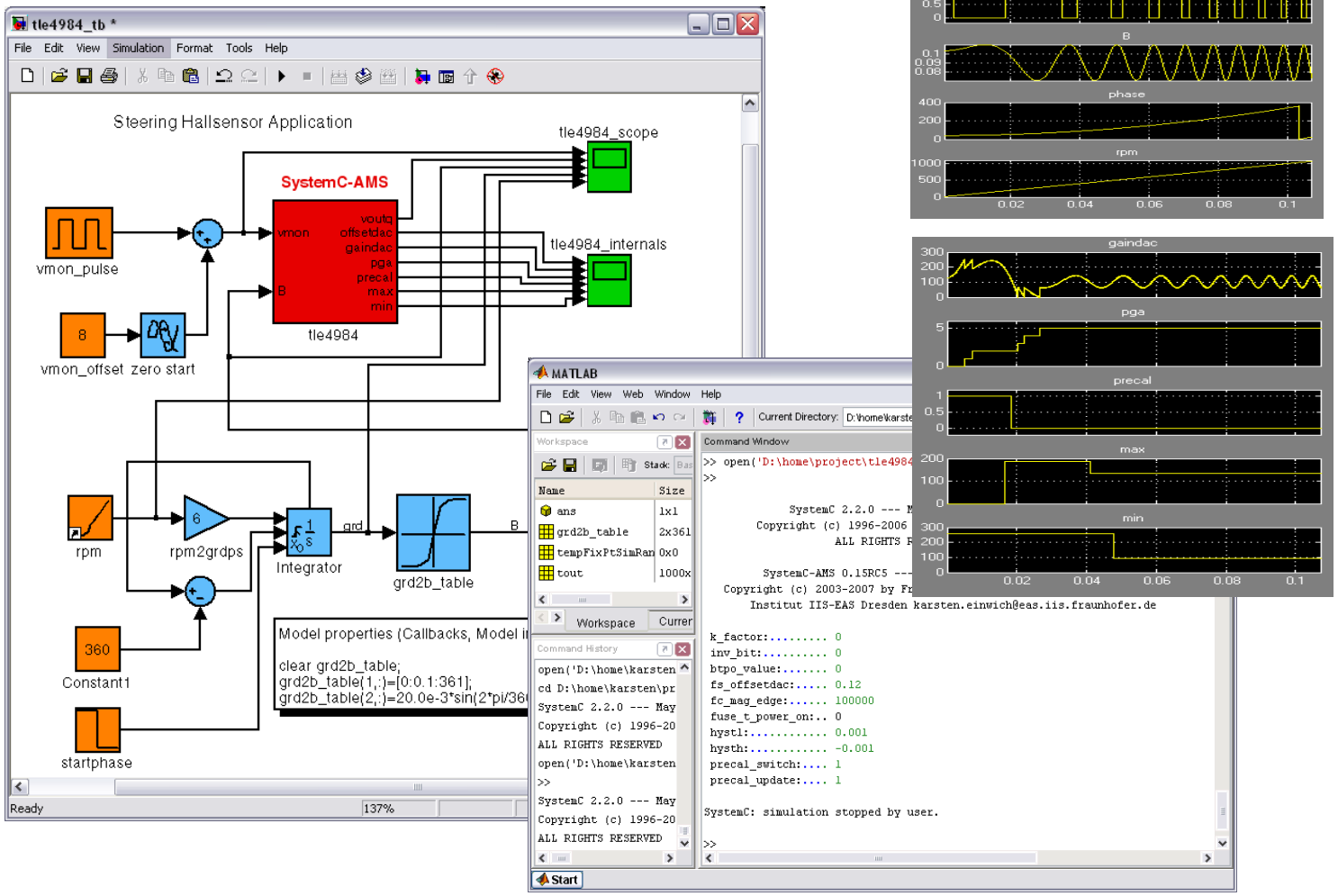
Block Level and/or short test sequences, block level verification

SystemC AMS Tools - COSIDE®



SystemC AMS - Model Export



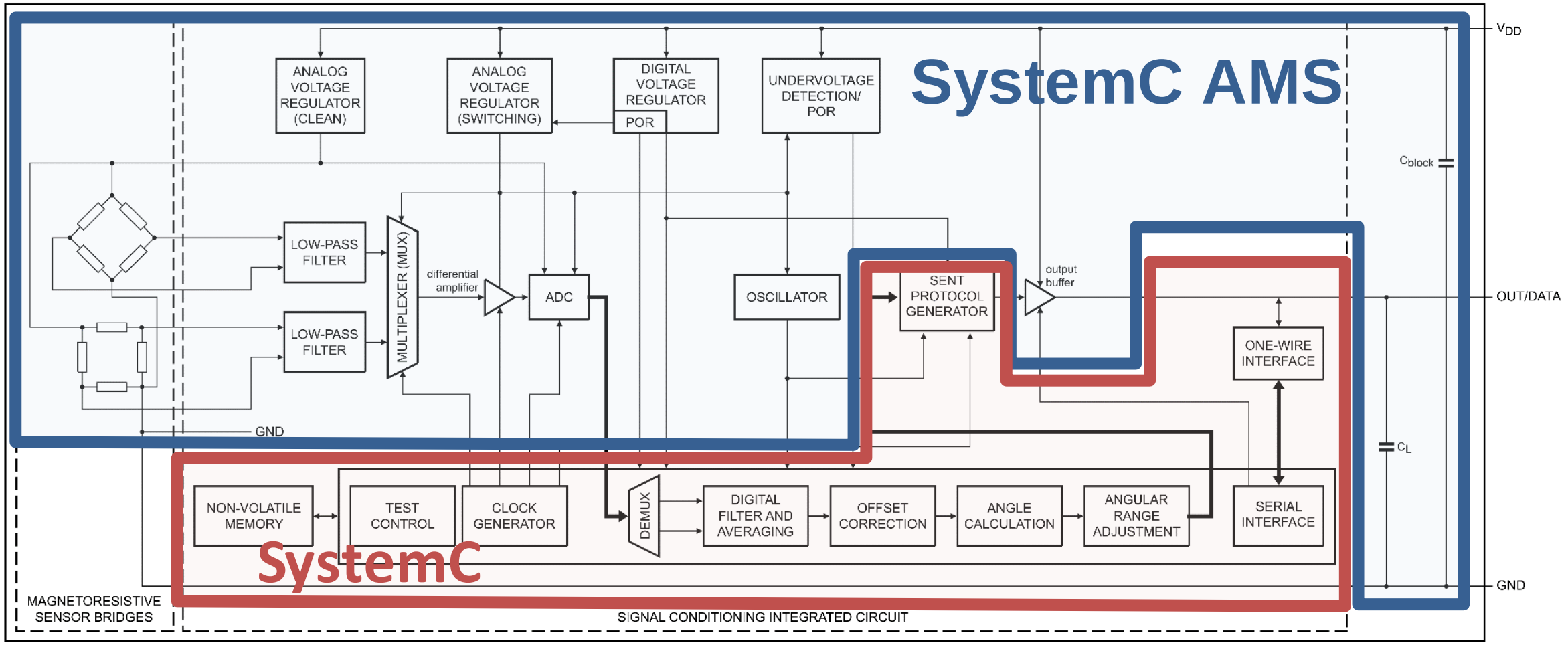


INDUSTRIAL APPLICATION

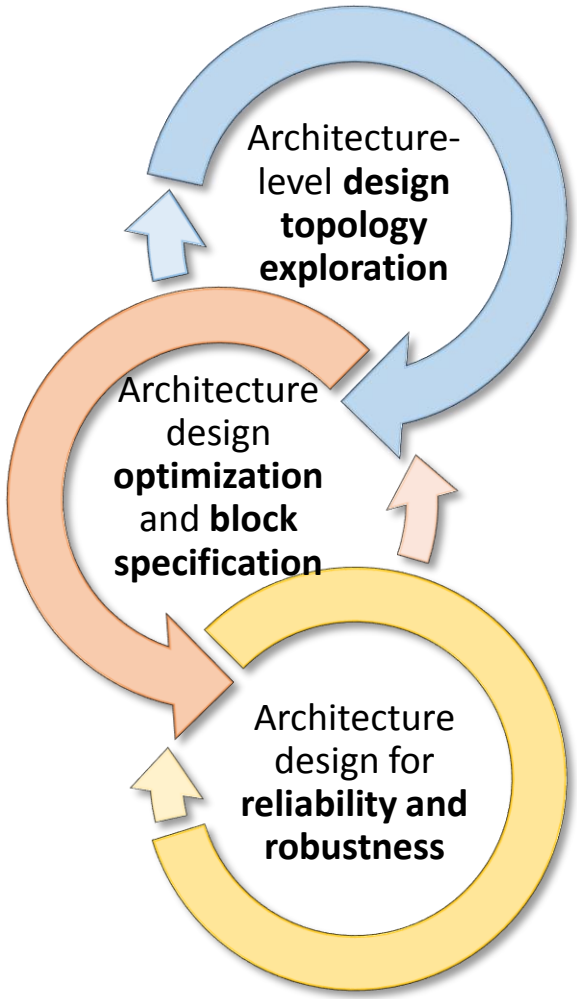
Application example: NXP MR Sensor application

- Magnetic angular sensor product with digital output
  - Magneto Resistive (MR) sensor bridges
  - AMS signal processing: Filters, ADC, oscillator, voltage regulators,…
  - Digital signal processing: Angle calculation (CORDIC), OWI interface, SENT protocol, Memory, FSM, …
  - Integrated blocking and load capacitors
- Demanding automotive application area
  - Position of throttle, pedal position, active suspension, electronic steering, ...
- Primary design and verification use case: concept development
  - Architecture-level design topology exploration
  - Architecture design optimization and design centering
  - Architecture design for reliability and robustness

Functional and architecture diagram



ESL design refinement methodology

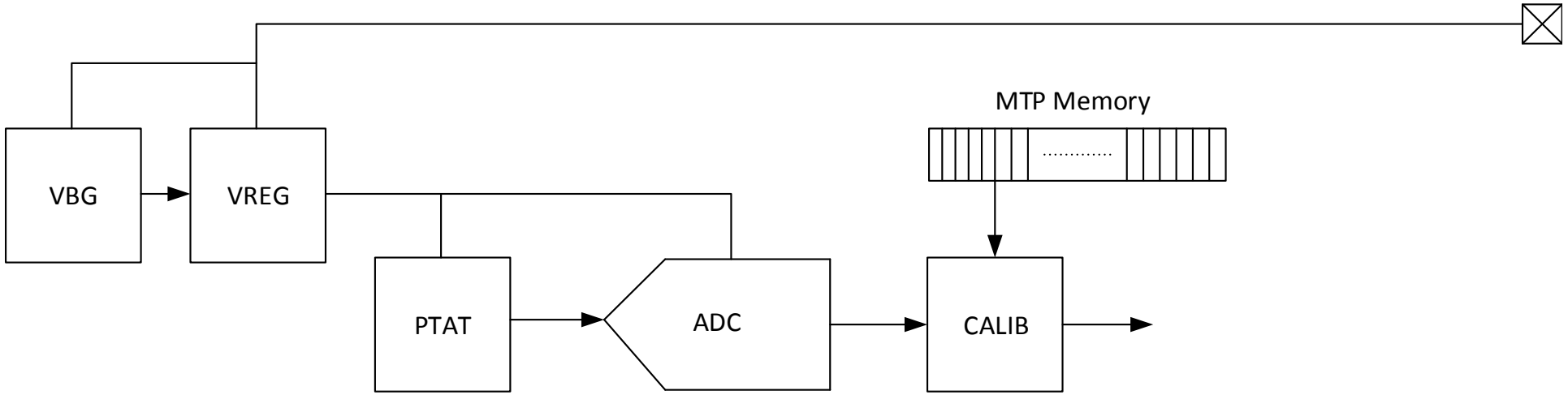


- Implement the architecture in SystemC AMS and SystemC
- Design refinement using transfer functions, switches, accurate regulation and control loops

- Modeling of passive components, thermal noise, 1/f noise and non-idealities
- Include characterized behavior in the system-level model, incl. PVT variations and dependencies

- Monte Carlo (MC) simulations using statistical library
- Apply extensive failure injection and analysis
- If MC or failure analysis fails, re-optimize system architecture

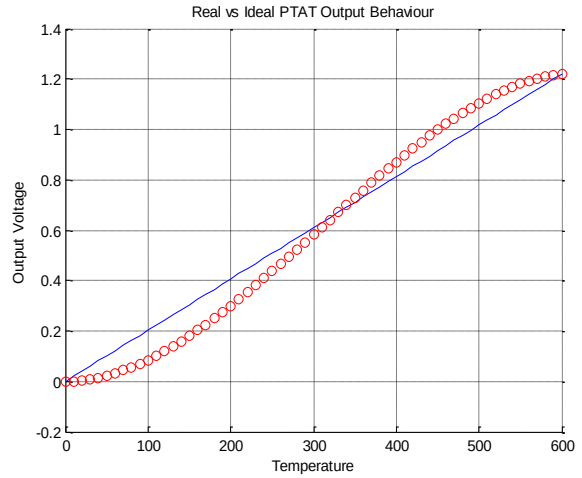
Temperature sensor in SystemC AMS

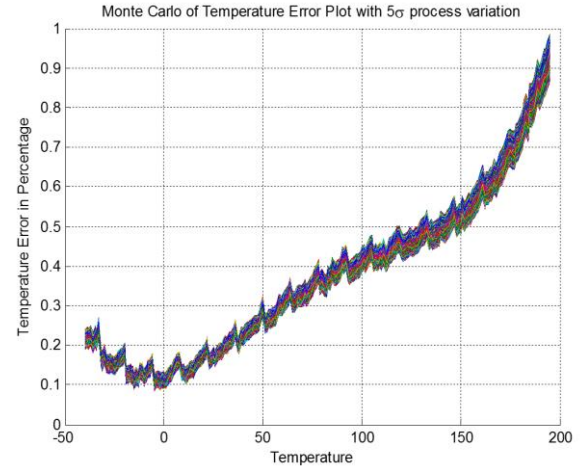


- Modeling objectives
  - Develop system-level model of temperature measurement and linearization part of the sensor
  - Start with abstract functional model, then refine the model based on characterization data from circuit-level implementation

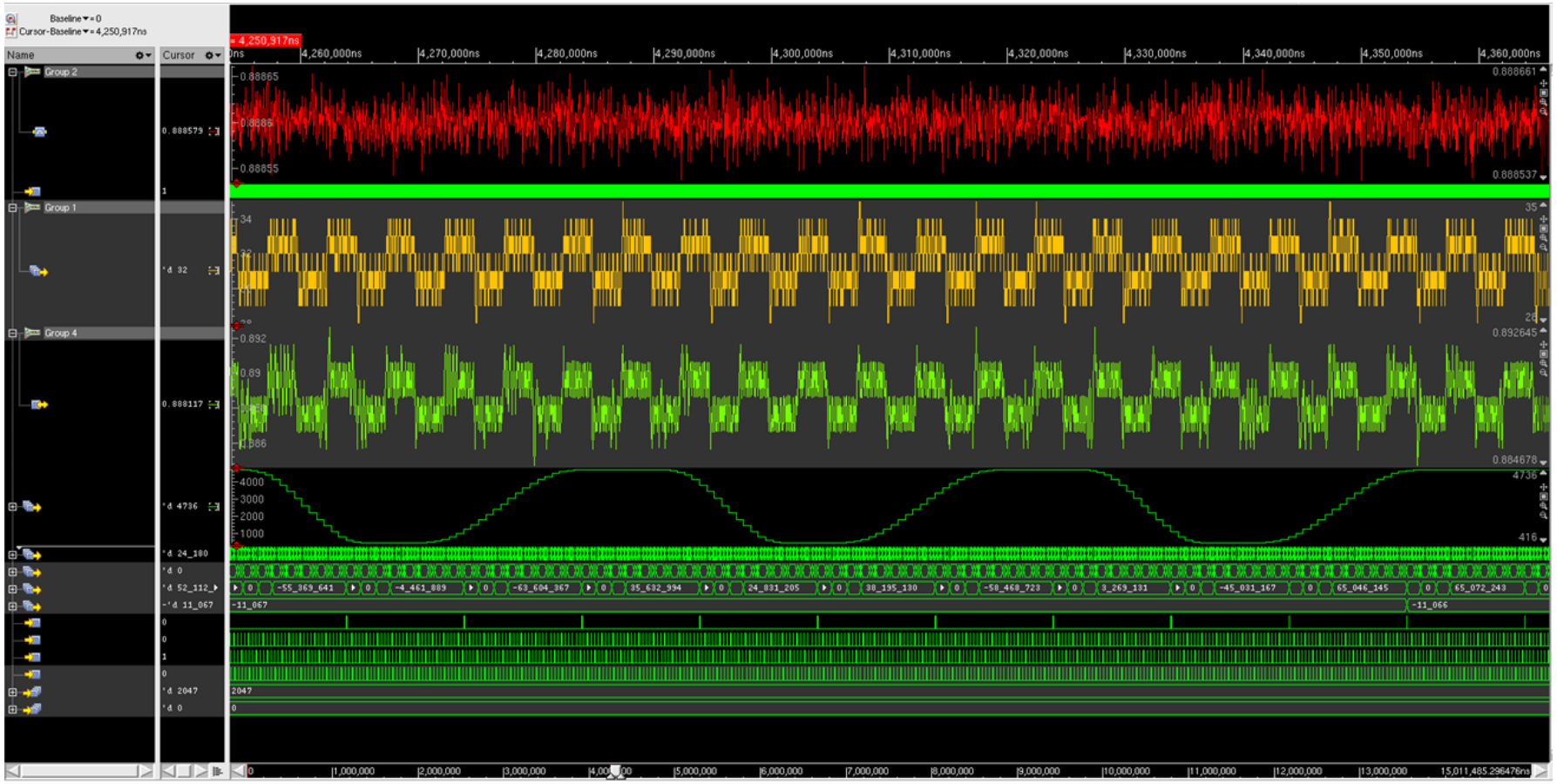
Temperature sensor refinement process

- Phase 1: Architecture-level design topology exploration
  - Algorithm level model for digital and transfer functions for AMS
  - PTAT modeled using theoretical nonlinear equation, ADC modeled with transfer function and a quantizer. The CALIB block is then developed to correct the nonlinearity.
- Phase 2: Architecture design optimization and block specification
  - More refined AMS models with non-idealities are introduced
  - This phase is budgeting extraction of specification for every block
- Phase 3: Architecture design for reliability and robustness
  - AMS system-level models are dimensioned based on the results of circuit analysis
  - Perform 5σ Monte Carlo (MC) using statistical library (GSL)





Mixed-signal transient simulation



- The SystemC AMS simulation kernel & models are compiled against the SystemC API available in 3rd party tools
- Interactive tracing and debug of SystemC AMS signals is supported by using the vendor API

Typical simulation speed

Design refinement step	Simulation speed	
	Simulation time (sec)	Wall clock time (sec)
Phase 1: Architecture-level design topology exploration	1	~60
Phase 2: Architecture design optimization and block specification	1	~500
Phase 3: Architecture design for reliability and robustness (using Monte Carlo)	1ms	~5 (per MC run)

Conclusions

- Application of SystemC AMS in a MR Sensor application
  - Heterogenous system containing digital and analog signal processing functionality
  - SystemC AMS used to mix (digital) algorithm with analog functionality
  - Efficient system simulations including Monte Carlo analysis
- SystemC AMS offers unique modeling features for mixed-signal system-level concept and architecture design
  - Supports different levels of abstraction for design refinement (TDF, LSF and ELN) to mix-and-match abstraction, accuracy and simulation speed
  - C++ libraries (e.g. BOOST and the GNU Scientific Library) can be added for dedicated tasks (e.g. Monte Carlo analysis)

Questions

Finalize slide set with questions slide