FDL’04 Tutorial
September 16, 2004

Analog and Mixed-Signal System Design
with SystemC

Christoph Grimm
University Frankfurt
Germany
grimm@ti.informatik.uni-frankfurt.de

Karsten Einwich
Fraunhofer IIS – EAS Dresden
Germany
karsten.einwich@eas.iis.fhg.de

Alain Vachoux
EPFL
Switzerland
alain.vachoux@epfl.ch
Outline

♦ Introduction

♦ Part 1: Design and modeling issues for complex heterogeneous systems
  • What is a complex heterogeneous system, application fields, motivations
  • Modeling strategies for AMS systems

♦ Part 2: Using SystemC for AMS systems
  • Introduction to SystemC 2.0 & use for AMS systems
  • Proposed AMS extensions

♦ Part 3: Application examples
  • Electronic example: PLL
  • Automotive example: PWM driver
  • Telecommunication example: xDSL

♦ Conclusions
Introduction: Tutorial Objectives

♦ To review design and modeling issues for complex heterogeneous systems

♦ To present a prototype implementation of modeling and simulation of mixed discrete/continuous systems in SystemC

♦ To provide some typical application examples
Part 1

Emerging Application fields (1/2)

♦ Ambient Intelligence Systems:
  • Hardware (IPs, Cores)
  • Software (Megabytes!)
  • Analog components:
    Converters, PLL, …
    Sensors
    RF/Wireless

♦ Automotive Systems 20XX:
  • Hardware (IPs, Cores)
  • Software (Megabytes!)
  • Converters, Sensors
    Power electronics
    Mechanical components
    Maybe RF/Wireless
  • High Reliability+Safety!
Emerging Application fields (2/2)

♦ Design of future applications has to consider interactions between:
  • Digital Hardware
  • Analog Components
  • Software

♦ Complex heterogeneous systems are superset of A/D/S + environment

♦ Co-simulation with physical environment:
  • Virtual prototyping replaces “breadboards”
  • Virtual testbenches complement “synthetic” testbench
Mixed Discrete/Continuous Systems

- Mixed Discrete/Continuous (MDC) systems exhibit a mix of:
  - Discrete-event or discrete-time behaviors
  - Continuous-time behaviors

- Compared with Analog and Mixed-Signal (AMS) systems:
  - MDC are often far more complex:
    Converters, PLL, etc. are rather small components of an MDC
  - Coupling A/D can be modeled in a more simple, and thereby more efficient way, e.g. in discrete time steps
  - MDC can be more abstract, and also embrace a large fraction of software
HDL Use

♦ Can we use HDLs for modeling, design and verification of complex, heterogeneous systems?

♦ Radio Eriwan's answer is: **Yes, but …**

…Modeling megabytes of software in VHDL/Verilog, or integration thereof using CLI might not be very comfortable

…Simulation performance would be orders of magnitudes too slow (Grimm et al. @ FDL’01: Virtual Test-Drive of Anti-Lock brake system would take YEARS)
HDL Use

♦ Can we use HDLs for modeling, design and verification of complex, heterogeneous systems?

♦ A more helpful answer is: Yes, but …

… Use **SystemC** for modeling hardware/software systems

… Use abstract, behavioral models

… Use application + abstraction specific means for simulation and coupling of simulators
How Can a Modeling Language Help?

♦ With appropriate properties we can more easily specify models, and analyze properties of a model while ignoring many implementation issues.

---

**Part 1**

DE model: We need explicit synchronisation, events.

DF model: Synchronisation implicit, static scheduling.

♦ The use appropriate modeling properties is the key to:
  - Abstract modeling
  - Efficient simulation
Definition

A model of computation defines a set of rules that govern the interactions between model elements, and thereby specify the semantics of a model.

Remarks:

♦ A model of computation can also be seen as a formal, abstract definition of a machine that executes a class of models (executable model).

♦ A model of computation is independent from a graphical or textual language, which specifies the syntactical composition of model elements.
• Whether a model of computation is appropriate for a modeling issue depends on:

  • **Application, e.g.:**
    Control systems → time domain, nonlinear, asynchronous behavior
    RF systems → linear, static non-linearities, constant time steps

  • **Implementation, e.g.:**
    → Analog: netlist, Digital: DE, Software: UML

  • **Level of abstraction, e.g.:**
    → Digital: Transactions, Register transfer, Netlist, …

• Modeling of heterogeneous systems at different levels of abstraction requires the use and combination of different modeling platforms
Model Facets

♦ Interface:
  • I/O ports, communication protocols, parameters

♦ Behavior:
  • I/O relationships, algorithms, data flows, processes, states, equations, hierarchy

♦ Structure:
  • Topological organization, connectivity, hierarchy

♦ Geometry:
  • Shapes, dimensions, part assemblies

♦ Properties:
  • timings, power consumption

♦ Operating conditions:
  • Temperature, pressure, noise, mechanical stress, …
# Abstraction

## Time

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock ticks (synchronous syst.)</td>
<td>Discrete (integer value f(MRT))</td>
</tr>
<tr>
<td>Continuous (real value)</td>
<td></td>
</tr>
</tbody>
</table>

## Behavior

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Causal</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Discrete</td>
<td>Continuous/Signal flow</td>
</tr>
<tr>
<td>Continuous/Conservative</td>
<td></td>
</tr>
</tbody>
</table>

## Data

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Tokens ((un)interpreted)</td>
<td>Enumerated (symbols, alphabet)</td>
</tr>
<tr>
<td>Logic values</td>
<td>Integer values</td>
</tr>
<tr>
<td>Real values</td>
<td></td>
</tr>
</tbody>
</table>

## Primitives

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor, memory, bus, RF emitter/receiver, PLL, sensor, actuator</td>
<td>ALU, register, control, converter, filter, VCO</td>
</tr>
<tr>
<td>Logical gates, Op-Amp</td>
<td>Transistor, R, C, source</td>
</tr>
</tbody>
</table>
MoCs for mixed continuous/discrete systems

♦ Continuous-Time Signal-Flow MoC:
  • Requirements engineering, executable specifications
  • Example: Simulink block diagrams

♦ Timed Synchronous (Multirate) Dataflow MoC:
  • DSP algorithms
  • Examples: SPW (Coware), System Studio (Synopsys)

♦ Discrete Event MoC:
  • Digital realization at different levels of abstraction
  • Example: SystemC

♦ Continuous-Time Conservative MoC:
  • Analog circuits
  • Example: SPICE
Based on Process Networks

- Networks of concurrent processes (actors), or actors, communicating through unidirectional unbounded FIFO channels (arcs)

- Tokens represent data as atomic and usually uninterpreted elements

- Processes map input tokens onto output tokens

- A process fires (resumes) when enough tokens are available at its input:
  - Consumes input token(s) (blocking read)
  - Possibly computes a new internal state
  - Produces output token(s) (non-blocking write)

Untimed MoC
Number of consumed/produced tokens is constant for a process
- Static scheduling of processes
- Complete cycle: processes may be fired a finite number of times before returning to original state

Single-rate (or homogeneous) SDF
- Tokens are consumed/produced one at a time (ex.: adders, multipliers)

Multi-rate SDF
- Tokens are consumed/produced at various rates (ex.: decimators, interpolators, block (de)coders)
Useful for modeling digital signal processing systems

- Ideal DSP behavior
- Tokens = data samples
- Sampling rates are rationally related
- Step size between samples implicitly related to some global clock

EDA tools:
- Ptolemy II (Univ. Berkeley)
- SPW (Coware)
- System Studio (Synopsys)

Languages:
- LUSTRE, SIGNAL
- Esterel
- SystemC
Timed Synchronous Dataflow MoC

♦ Cosimulation between DSP, analog and RF domains

♦ Common representation of signals (arcs):
  - Frequency info: carrier frequency $f_c$
  - Time (baseband) info: in-phase component $I(t)$, quadrature component $Q(t)$, $t$

♦ Added attributes:
  - One time step and frequency carrier attached to each arc
  - Processes fired at constant rate
  - Optional I/O impedances

♦ Time steps and freq. carriers for each arc are computed by propagation algorithms

♦ EDA tool:
  Agilent Ptolemy

J.L. Pino, K. Kalbasi,
*Cosimulating Synchronous DSP Applications with Analog RF Circuits*,
Proc. IEEE Asilomar Conference on Signals, Systems, and Computers,
Discrete Event MoC

♦ Also based on process networks, but with a different communication mechanism:
  • Sequence of events in time
  • **Time**: integer multiple of some base time or real time
  • **Event**: (time stamp, value)

♦ Data: tokens, enumerated symbols, logical values, numerical values

♦ Dynamic scheduling of processes
  • Causality and determinism ensured through **delta delay iterations** or through extraction of **data dependencies**

♦ Main application: Concurrent hardware systems

♦ EDA tools:
  • Ptolemy II (Univ. Berkeley)
  • SystemC tools
  • VHDL(-AMS)/Verilog(-AMS)/SystemVerilog tools
Continuous-time MoC

♦ Based on Ordinary Differential Equations (ODEs) or Differential Algebraic Equations (DAEs)

♦ Signals are analytical functions of time
  • Piecewise differentiable segments
  • Real valued time

♦ Many methods to set up and to solve the system of equations:
  • Equation formulation methods (Nodal, Modified Nodal, Tableau, etc.)
  • Numerical methods (num. integration, NR linearization, linear sys. solver)
  • Symbolic methods

♦ EDA tools:
  • Ptolemy II (Univ. Berkeley)
  • Matlab/Simulink (MathWorks)
  • SPICE variants
  • VHDL-AMS/Verilog-AMS tools

♦ Applications:
  • Analog electrical systems
  • Physical systems (e.g. mechanical)
  • RF/microwave
  • Control systems

\[
y(t) = f(x(t), t) \\
\dot{x}(t) = f(x(t), u(t), t) \\
f(\dot{x}(t), x(t), u(t), t) = 0
\]
CT MoC: Signal-Flow/Block Diagrams

♦ Most abstract representation of physical/analog behavior
  • Non conservative behavior

♦ A SF model represents a computational structure as a directed graph
  • Arcs are transfer functions between CT signals (nodes)
  • Differential relations are expressed as their equivalent discrete formulation

♦ Block diagrams are dual representations of single port SF graphs
  • SFG (resp. BD) path => BD (resp. SFG) node

![Diagram of a control system block diagram](image)
CT MoC: Conservative Models

♦ More detailed representation of physical/analog behavior

♦ A conservative model represents the topology of the modeled system
  • Electrical systems: Kirchhoff’s networks meeting Kirchhoff’s laws (KCL, KVL)
  • Other physical systems: Generalized versions of KN and KCL/KVL laws
  • Bond graphs

♦ Netlist based models:
  • Topological connection of primitive elements
  • Elements defined by constitutive equations

♦ Two characteristic quantities:
  • Across (effort, e.g. voltage)
  • Through (flow, e.g. current)
CT MoC: Macromodels

♦ Simplified equivalent circuit or simplified system of equations that represents the I/O behavior

♦ Goal is to achieve fast simulation while keeping an acceptable level of accuracy

♦ Macromodel development techniques:
  • Circuit simplification
    Remove circuit elements, use simpler models
  • Circuit build-up
    Use ideal primitive elements
    Progressively add non-ideal behavior
  • Symbolic manipulations of circuit equations
Mixing Different MoCs

♦ Objective is to deal with system heterogeneity

♦ Hybrid MoC: Composition of control (FSM) and CT

♦ Mixed-signal or mixed discrete/continuous MoC: Composition of DE and CT

♦ MoCs are usually combined using a hierarchical approach

♦ Interaction semantics define how semantic properties of interacting MoCs are related to each other

♦ Time is the most critical interacting property
  • Untimed DF and timed DE
  • Timed DE and timed CT
(S)DF in DE

- DF subsystems appear as zero-delay blocks

- Each activation of a DF block must perform a complete cycle

- DF subsystem may be over-constrained:
  - Can only fire when all of its inputs have an event
  - Alternative: generate the needed data using the most recently updated value

- Multi-rate SDF block:
  - A single event at inputs may not be enough to activate the whole block
  - More than one token may be produced at the output (time stamp?)

Example: VHDL-AMS initialization and time-domain simulation cycle

- MoCs interact as peers (no hierarchy)

Diagram:

- Computation of analog solution points $-> t = T_n' \leq T_n$
- $T_c := \min(T_n, T_n')$
- $T_c = \text{time}'\text{high}$
  - yes $ightarrow$ End of simulation
  - no
    - Signal updates
    - Execution of all processes sensitive to signal updates
    - Computation of next time $T_n$
      - yes $\rightarrow T_n = T_c$
      - no

Part 2: Using SystemC for AMS Systems

♦ Overview of SystemC 2.0
  • Why C based design?
  • SystemC approach and use flow
  • Simple examples using the core language

♦ Modeling AMS systems with SystemC 2.0
  • Discrete-event modeling of continuous-time behaviors
  • Representation of linear dynamic systems
  • Adaptative time step approach

♦ Proposed SystemC AMS extensions
  • Architecture of the extensions
  • Language constructs, class definitions
Why C based design? (1/2)

- Co-design of hardware/software systems
  - C/C++/UML provide means for modeling software
  - HDLs provide means for modeling hardware

Diagram:

- Software development
  - C, C++, UML, ...
  - Re-partitioning requires translation
  - SW developers need C/C++ models

- Hardware development
  - SystemVerilog, VHDL, Verilog

System design
Why C based design? (2/2)

♦ Pragmatic approach: Use C/C++/UML for HW/SW system design

```
Diagram:

System design

Software development
C, C++, UML, ...
+ means for modeling timing, concurrency and signal types

Hardware development
SystemVerilog, VHDL, Verilog
```

FDL’04 tutorial, Sept. 16 2004
Analog and Mixed-Signal System Design with SystemC
### The SystemC Approach

- **SystemC** is C++ plus a class library to support system-level HW modeling

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard Channels for Various Models of Computation</strong></td>
<td>Kahn Process Networks, Static Dataflow, etc.</td>
</tr>
<tr>
<td><strong>Methodology-Specific Channels</strong></td>
<td>Master/Slave Library, etc.</td>
</tr>
<tr>
<td><strong>Elementary Channels</strong></td>
<td>Signal, Timer, Mutex, Semaphore, FIFO, etc.</td>
</tr>
<tr>
<td><strong>Core Language</strong></td>
<td>Modules, Ports, Processes, Events, Interfaces, Channels, Event-Driven Simulation Kernel</td>
</tr>
<tr>
<td><strong>Data-Types</strong></td>
<td>4-valued logic types (01xz), 4-valued logic-vectors, Bits and bit-vectors, Arbitrary-precision integers, Fixed-point numbers, C++ user-defined types</td>
</tr>
<tr>
<td><strong>C++ Language Standard</strong></td>
<td></td>
</tr>
</tbody>
</table>
Part 2

SystemC w.r.t. other Design Languages

Requirements

Architecture

HW/SW

Behavior

Functional Verification

Test bench

RTL

Gates

Transistors

Verilog

VHDL

System Verilog

Vera

Sugar

SystemC

Matlab
SystemC Use Flow

- **SystemC model**
- **SystemC library**
- **C++ compiler and linker**
- **Executable code (simulator)**
- **C++ debugger**
- **Waveform viewer**
Part 2

Architecture of a SystemC 2.0 Model

♦ Separation of behavior and communication

| ♦ Communication refinement: Channel's behavior may change from very abstract (e.g. transactions, protocol) to very detailed (e.g. hardware signals) without requiring to change module's behaviors |

[Diagram showing separation of behavior and communication in SystemC 2.0 Model]
Part 2  

**SystemC Core Language: Modules and Ports**

- Structural units are called **modules**

- Modules are inherited from the class *sc_module*
  - Macro *SC_MODULE* does the job for you

- Modules communicate with environment via **ports**

- Ports are instances of the classes (where *T* denotes a data type):
  - *sc_in*T or *sc_out*T or *sc_inout*T

- Ports are declared in the general form *sc_port*<class IF, int N=1>*
  - IF = interface (see later)

```c++
SC_MODULE(my_module)
{
    sc_in<type> input;
    sc_out<type> output;

    // C++ methods here (behavior)
    SC_CTOR(my_module)
    {
        // C++ code here (initialization)
    }
};
```
Behavior of modules is described by discrete processes.

Processes are defined as C++ methods that must be registered to the simulation kernel by the following macros:

- `SC_THREAD(method_name)`
- `SC_METHOD(method_name)`

Processes are activated by events which are specified in a sensitivity list following registration of the process:

- `sensitive[_pos|_neg] (<< [signal|event])*;`

```cpp
#include "systemc.h"

SC_MODULE(add)
{
    sc_in<int> in1;
    sc_in<int> in2;
    sc_out<int> outp;

    void do_add()
    {
        outp = in1 + in2;
    }

    SC_CTOR(add)
    {
        SC_METHOD(do_add);
        sensitive << in1 << in2;
    }
};
```
SystemC Core Language: Signals & FIFOs

- Modules communicate via channels

- Channels are accessed via interfaces
  - An interface defines a set of abstract methods that can be used for communication
  - Processes use interface methods `read()`, `write(...)`, `event()`, ...

- Signals are a class of primitive channels that model hardware signals
  - Class `sc_signal<T>` defines the implementations of abstract interface methods
  - Port `sc_in<T>` is derived from `sc_port<sc_signal_in_if<T>,1>`

- FIFOs are another class of primitive channels that model bounded FIFO queues
  - Class `sc_fifo<T>` defines the implementations of abstract interface methods
  - Port `sc_fifo_in<T>` is derived from `sc_port<sc_fifo_in_if<T>,1>`
Hierarchical Model Example

```c
#include "systemc.h"
#include "adder.h"
#include "latch.h"

SC_MODULE(dut) {
    sc_in<bool> clk;
    sc_in<int> in1, in2;
    sc_out<int> outp;

    sc_signal<int> internal_signal;

    adder* add1;
    latch* latch1;

    entity dut is
        port (signal clk: in bit;
                signal in1, in2: in bit;
                signal outp: out bit);
    end entity dut;

    architecture str of dut is
        signal internal_signal: bit;
        begin
            add1: entity work.adder(dfl)
                port map (in1 => in1,
                          in2 => in2,
                          outp => internal_signal);
            latch1: entity work.latch1(bhv)
                port map (clk => clk,
                          inp => internal_signal,
                          outp => outp);
        end architecture str;

    SC_CTOR(dut) {
        add1 = new adder("add1");
        add1->in1(in1);
        add1->in2(in2);
        add1->outp(internal_signal);

        latch1 = new latch("latch1");
        latch1->clk(clk);
        latch1->inp(internal_signal);
        latch1->outp(outp);
    }
};
```
Part 2

Testbench Example

```c
#include "dut.h"
#include "stimuli_generator.h"

int sc_main(int argc, char* argv[]) {
    sc_signal<int> signal1, signal2, signal3;
    sc_clock clock1("clock1", 1.0, SC_US);
    stimuli_generator stg1("stg1");
    stg1.sig1(signal1);
    stg1.sig2(signal2);
    dut dut1("dut1");
    dut1.inp1(signal1);
    dut1.inp2(signal2);
    dut1.out(signal3);
    dut1.clock(clock1);
    ...
    sc_trace_file *tf sc_create_vcd_trace_file("simplex");
    sc_trace(tf, clock1, "clock1");
    sc_trace(tf, signal1, "in1");
    sc_trace(tf, signal2, "in2");
    sc_trace(tf, signal3, "out");
    sc_trace(tf, dut1.internal_signal, "dut_signal");
    sc_start();
    sc_close_vcd_trace_file(tf);
    return(0);
}
```
Modeling analog modules using discrete-event SystemC

1. Split the equation system in non-conservative (directed) connected blocks/modules

2. Model the behavior of the blocks in a way that they embed his own solver

3. Use a SystemC-MoC to solve the overall equation system
Limitations

♦ Modules can be connected by non-conservative signals only

♦ No global view to the overall equation system – a non-solvable systems can’t be detected

♦ Loops of connected modules must (should) have a delay

♦ The system decomposition is influenced by the non-conservative signal limitation – it will not be always possible to provide general models and it can be difficult to understand the model

♦ The modeling effort depends on the block and can be very high
Modeling analog modules with the discrete-event SystemC

1. Split the equation system in non-conservative (directed) connected blocks/modules

2. Model the behavior of the blocks in a way that they embed his own solver

3. Use a SystemC-MoC to solve the overall equation system
Split the equation system into non-conservative modules

❖ Some guidelines

• Model the (black box) behavior of the interested values – use your system knowledge

• If possible consider an output resistance as zero and/or the following input resistance as infinite

• Split the wires into directed signals which are carry the current or the voltage

• Try to split into linear dynamics and non-linear static's

• Split into control and signal flow
Modeling analog modules with the discrete-event SystemC

Principle:

1. Split the equation system in non-conservative (directed) connected blocks/modules

2. Model the behavior of the blocks in a way that they embed his own solver

3. Use a SystemC-MoC to solve the overall equation system
Model the modules in a way that they embed the solver

```systemc
SC_MODULE(kv2w)
{
    sc_quantity_in v2w;
    sc_quantity_out vtr;

    // control de - inport
    sc_in<double> k_v2w;

    void sig_proc();

    SC_CTOR(kv2w)
    {
        SC_THREAD(sig_proc);
    }

    void kv2w::sig_proc()
    {
        while(true) {
            double v2w_tmp=v2w.read();
            double vtr_tmp;

            vtr_tmp=k_v2w.read() * vtr_tmp;

            vtr.write(vtr_tmp);
        }
    }
}
```
Part 2

**Modeling linear analog dynamic behavior**

- **Example: RC low pass**

\[ H(s) = \frac{Y(s)}{U(s)} = \frac{1}{RCS + 1} \]

\[ u(t) = y(t) + RC \frac{dy}{dt} \]

\[ \frac{dy}{dt} \approx \frac{y(t_n) - y(t_{n-1})}{t_n - t_{n-1}} \]

\[ y(t_n) = \frac{(t_n - t_{n-1})u(t_n) + RCy(t_{n-1})}{(t_n - t_{n-1}) + RC} \]

```systemc
SC_MODULE(low_pass) {
    sc_quantity_in u;
    sc_quantity_in y;
    double TAU; // time constant
    double state; // internal state

    void sig_proc()
    {
        sc_time DT(10, SC_US);
        double DDT = DT.to_seconds();
        while (true)
        {
            state = (state*TAU + u.read()*DDT) / (TAU + DDT);
            y.write(state);
        }
    }

    SC_CTOR(low_pass)
    {
        // initializations
        TAU = 2.0e-4; state = 0.0;
        // register thread
        SC_THREAD(sig_proc);
    }
};
```
“Analog” Representation of linear dynamic Systems

♦ Transfer function

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

♦ Easy extraction from networks, ...

♦ Zero-Pole representation

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

♦ Operational amplifier, analog filters

♦ State Space equations

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

♦ Good state control
Transformation to Discrete Time (1/2)

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

MATLAB Code:

```matlab
[bz,az]=bilinear(b,a,FS);

#discrete filter identification
#weigthing vector
wt=[1:-1/size(w,2):1/size(w,2)];

[bz,az]=invfreqz(hs,w/FS,2,2,wt,100,0.001);

\[ x[n+1] = \tilde{A}x[n] + \tilde{B}u[n] \]
\[ y[n] = \tilde{C}x[n] + \tilde{D}u[n] \]

\[ y = \frac{1}{\tilde{a}_0} ((\tilde{b}_n z^{-n} + \tilde{b}_{n-1} z^{-n-1} + \ldots + \tilde{b}_0) u - (\tilde{a}_m z^{-m} + \tilde{a}_{m-1} z^{-m-1} + \ldots + \tilde{a}_1 z^{-1}) y) \]
Solving of discrete Time System Representation

♦ Transfer function

\[ H(z) = \frac{bd_n \cdot z^{-n} + bd_{n-1} \cdot z^{n-1} + \ldots + bd_0}{ad_m \cdot z^{-m} + ad_{m-1} \cdot z^{m-1} + \ldots + ad_0} \]

♦ State Space

\[ x[n+1] = Ad \cdot x[n] + Bd \cdot u[n] \]
\[ y[n] = Cd \cdot x[n] + Dd \cdot u[n] \]

```cpp
void hz::sig_proc()
{
   //straightforward implementation
   //input shift register
   for(i=0;i<n;i++) zu[i+1]=zu[i];
   zu[0]=u;

   //calculate nominator
   for(i=0,y=0.0;i<=n;i++) y+=bd[i]*z[i];

   //calculate denominator
   for(i=1;i<=m;i++) y-=ad[i]*zy[i];
   y=y/ad[0];

   //y-shift register
   for(i=0;i<m;i++) zy[i+1]=zy[i];
   zy[0]=y;
}
```

\[ y = \frac{1}{ad_0} \left( (bd_n \cdot z^{-n} + bd_{n-1} \cdot z^{n-1} + \ldots + bd_0) \cdot u - (ad_n \cdot z^{-m} + ad_{n-1} \cdot z^{m-1} + \ldots + ad_{m-1}z^{-1}) \cdot y \right) \]
Modeling conservative Blocks

- Encapsulation into one block

- Transformation to a non conservative system

- Transform this system to a prepared system representation (transfer function, state space equations)
Modeling linear electrical Networks

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

\[ \dot{x} = Ax + Bu \]
\[ y = Cx + Du \]
**RC-low pass Example**

\[
\frac{V_{out}}{V_{in}}(s) = \frac{1}{Cs} \cdot \frac{1}{R + \frac{1}{Cs}} = H(s)
\]

\[
H(s) = \frac{1}{1 + RCs}
\]

\[
b[0] = 1
\]

\[
a[0] = 1
\]

\[
a[1] = R \cdot C
\]

```cpp
SC_MODULE(rc_lp)
{
  sc_quantity_in  Vin;
  sc_quantity_out Vout;

  SC_HAS_PROCESS(rc_lp);
  rc_lp(sc_module_name nm,
        double R,    //parameters
        double C,
        sc_time Ts   // sample period
        ):sc_module(nm),a(2),b(1)
  {
    SC_THREAD(time_step);
    ts=Ts;
    b[0]=1.0; a[0]=1.0; a[1]=R*C;
  }
  void time_step()
  {
    while(true)
    {
      Vout.write(Ltf(b,a,s,ts,id,Vin.read()));
    }
  }
  private:
  vector<double> a, b, s;
  LTF_ID    id;
  sc_time   ts;
}
```
**Complex Example**

![Complex Example Diagram]

\[
A = \begin{bmatrix}
\frac{2rb + r1 + r2}{2c2rb + c2r1r2} & 1 & 1 \\
\frac{1}{2c2rb + c2r1} & \frac{1}{2cbrb + cbr1} & \frac{1}{2cbrb + cbr1} \\
\frac{2rb + r2}{4rb + r2} & \frac{4rb + r2}{4rb + r2} & \frac{4rb + r2}{4rb + r2} \\
\frac{1}{2c2rb + c2r1} & \frac{1}{2cbrb + cbr1} & \frac{1}{2cbrb + cbr1} \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
r2 & 2rb + r1r2 & 0 \\
2rb & 4rb + r2 & 2rb + r1r2 \\
4rb + r2 & 2rb + r2 & 4rb + r2 \\
4rb + r2 & 4rb + r2 & 4rb + r2 \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
1 & 0 & 1 \\
2c2rb + c2r1 & 0 & 0 \\
2c2rb + c2r1 & 0 & 0 \\
\end{bmatrix}
\]

\[
D = \begin{bmatrix}
\frac{1}{2c2rb + c2r1} & 0 & 0 \\
\frac{1}{2cbrb + cbr1} & 0 & 0 \\
\frac{1}{2cbrb + cbr1} & 0 & 0 \\
\end{bmatrix}
\]

\[
[[v_tr,[Tip,Ring],vtr]
[r_b1,[Tip,1],rb]
[c_b1,[1,0],cb]
[r_b2,[1,A],rb]
[r_1,[A,2],r1]
[c_2,[2,3],c2]
[r_2,[2,3],r2]
[v_in,[3,B],vin]
[rb_3,[B,4],rb]
[c_b2,[4,0],cb]
\]

---

**Mathematica / Analog Insydes**

**SystemC**
Modeling of ideal Switching

- System representation has to be re-initialized

- State interpretation depends on the implementation

- The states must have values which remain constant during switching (e.g. the energy values of network components: charge of capacities, magnetic flux of inductivities)

- Using state space equations this can be realized in a simple way
Example for switching Network

\[
\begin{align*}
\dot{Q} &= -\frac{1}{C \cdot R} \cdot Q + \frac{1}{R} \cdot vin \\
vout &= \frac{(-1)}{C} \cdot Q + 0 \cdot vin
\end{align*}
\]

\[A = \frac{-1}{C \cdot R} \quad B = \frac{1}{R} \\
C = \frac{-1}{C} \quad D = 0\]

```c
void init()
{
    A1[0]=1.0/(C*R1);    A2[0]=1.0/(C*R2);
    B1[0] =-1.0/R1;      B2[0] =-1.0/R2;
    C1[0]=(-)1.0/C;      C2[0]=(-)1.0/C;
    D1[0]=0.0;          D2[0]=0.0;
}

void sig_proc()
{
    //state vector S will be hold
    if(ADSL_LITE) OUT=SS(A1,B1,C1,D1,S,id1,INP);
    else OUT=SS(A2,B2,C2,D2,S,id2,INP);
}
```
Modeling analog modules with the discrete-event SystemC

**Principle:**

1. Split the equation system in non-conservative (directed) connected blocks/modules

2. Model the behavior of the blocks in a way that they embed his own solver

3. **Use a SystemC-MoC to solve the overall equation system**
Dataflow Model of Computation

- Simple firing rule: A block is called if enough sample available at his inports
- A block reads (removes sample) from the inports and writes to the outports
- For synchronous dataflow this numbers of read/written samples are constant for all block calls
- The scheduling follows the signalflow direction

\[
\text{out} = f_3( f_2( f_1(\text{in}) ) )
\]
Solve the overall equation system using a SystemC - MoC

Synchronous Dataflow scheduling

• The sample period is smaller than two times of the smallest not negligible time constant of the system

• The signal is linear between sampling points

• The sample period is constant

• The (digital) sample time points equals to the analog calculation points
Static Dataflow Scheduling without a Loop

- The synchronous dataflow MoC determines execution order in signalflow direction
- All modules called at the same (SystemC) time point
- The time delay between modules is zero
- Occurrence of SystemC – trigger events determining the time step (e.g. constant time steps)

\[ \text{out} = f_3( f_2( f_1(\text{in}) ) ) \]
Static Dataflow Scheduling without a Loop

- The synchronous dataflow MoC determines execution order in signalflow direction.
- All modules called at the same (SystemC) time point.
- The time delay between modules is zero.
- Occurrence of SystemC – trigger events determining the time step (e.g. constant time steps).

Time : 0.1 s

\[ \text{out} = f_3( f_2( f_1(\text{in}) ) ) \]
Static Dataflow Scheduling with Loop

Time : 0.1

- Loops must have a delay to allow scheduling
- A delay inserts (writes) during the initialization phase a sample
- For analog modeling, the delay is a "hopefully" acceptable approximation

\[
\text{out} = f_1(\text{in}, f_2(\text{out})z^{-1})
\]
Static Dataflow Scheduling with SystemC

- Using the primitive channel sc_fifo

- Using OSCI dataflow modeling style (working group currently sleeping)

- Using user defined channel
**Synchronous Dataflow Scheduling with sc_fifo**

- Interface to sc_fifo – channel is blocking read and write

- Read from an empty fifo suspends the calling process until enough data are available which will be written by an other process

- Write to a full fifo suspends the calling process until an other process has read data from this fifo

- For single rate synchronous dataflow scheduling fifos of size 1 are used

- **Attention** – an outport can drive only one inport – a fork/splitter block is required, which copies a sample of the inport to multiple outports
SystemC 2.0 realization of a FIFO - Communication

```c
//recommendation
typedef sc_quantity sc_fifo<double>;
typedef sc_quantity_in sc_fifo_in<double>;
typedef sc_quantity_out sc_fifo_out<double>;

SC_MODULE(analog_block)
{
    sc_quantity_in in;
    sc_quantity_out out;

    void do_analog() {
        while(true) {
            // !!! one read and write
            // per time step only !!!
            double tmp_in=in.read();
            //do analog function
            out.write(tmp_out);
        }
    }

    SC_CTOR(analog_block)
    {
        SC_THREAD(do_analog);
    }

    sc_quantity conn1(1), conn2(1);
    const_source src1("sc1");
    src1.out(con1);
    src1.T=sc_time(1.0,SC_MS);
    src1.value=1.0;

    analog_block a1("a1");
    a1.in(con1);
    a1.out(con2);

    sink s1("s1");
    s1.in(con2);
}
```
Synchronization to discrete event signals

```systemc
SC_MODULE(sdf_de)
{
    sc_quantity_in ana_inp;
    sc_quantity_out ana_outp;

    //de-ports are connected
    //to sc_signal<type>
    sc_in<sc_logic> de_in;
    sc_out<sc_int<3> > de_out;

    SC_CTOR(sdf_de)
    {
        SC_THREAD(sdf_de);
    }
};
```

```c
void do_analog()
{
    while(true)
    {
        double tmp_in=ana_inp.read();

        if(de_in.read()==1)
        {
            //do something analog,
            //assign to tmp_out
        }
        else {.....}

        sc_int<3> de_outv= ???;
        de_out=de_outv;
        de_out.write(tmp_out);
    }
}
```
Approaches to use variable time steps

Example: 1st order lowpass filter

```c
#include "systemc.h"
#include "AnalogSys.h"

struct lp1 : sc_module, analog_module {
    sc_in<double> lp_in;  // filter input
    sc_out<double> lp_out; // filter output
    sc_out<double> dt_out; // own timestep
    double vin_thresh; // input variation threshold
    double vin_old;  // input value at preceding activation

    void field(double *var) const { // state derivative
        const double TAU = 2.0e-4;
        var[0] = (lp_in.read() – state[0])/TAU;
    }
    void sense() {
        double vin = lp_in.read();
        if (fabs(vin – vin_old) > vin_thresh) {
            activation.notify();
        }
        vin_old = vin;
    }
    void calculus() {
        state[0] = 0.0;
        while (true) {
            analog_module::step();
            lp_out.write(state[0]);
            dt_out.write(dt);
        }
    }
}; // lp1
```

```c
void lp1::field(double *var) const {
    const double TAU = 2.0e-4;
    var[0] = (lp_in.read() – state[0])/TAU;
}

void lp1::sense() {
    double vin = lp_in.read();
    if (fabs(vin – vin_old) > vin_thresh) {
        activation.notify();
    }
    vin_old = vin;
}

void lp1::calculus () {
    state[0] = 0.0;
    while (true) {
        analog_module::step();
        lp_out.write(state[0]);
        dt_out.write(dt);
    }
}
```
Using an Adaptative Time Step (3/3)

♦ Explicit numerical integration methods
  • e.g., Forward Euler, Adams-Bashforth

♦ Tuning of parameters required to achieve acceptable accuracy/CPU time
  • Input variation thresholds
  • Minimum/maximum time step
  • Time step multiplication factor
  • Tolerances (reltol, abstol)
SystemC-AMS

- Motivation
- Architecture
- Implementation
- Examples
Requirements

♦ Different and partial oppositional requirements

♦ A lot of very efficient however high specialized existing solutions

♦ A generic and extendable approach necessary

♦ The approach must be simple and efficient feasible

♦ The generic concept of SystemC has to be extended for AMS-Systems
SystemC-AMS Use Flow

SystemC-AMS library

SystemC model
SystemC - AMS Realization

♦ Analog Module
  • Container class for analog Ports and primitive behavior

♦ Analog Port
  • Provides access to an connected interface/channel

♦ Analog Interface
  • Provides access routines

♦ Analog Channel
  • Implements access routines
Part 2

The SystemC - AMS

- SystemC – AMS extents SystemC by putting on the core language

SystemC-AMS Extension

<table>
<thead>
<tr>
<th>Core Language</th>
<th>Data–Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules</td>
<td>4–valued logic types (01xz)</td>
</tr>
<tr>
<td>Ports</td>
<td>4–valued logic–vectors</td>
</tr>
<tr>
<td>Processes</td>
<td>Bits and bit–vectors</td>
</tr>
<tr>
<td>Events</td>
<td>Arbitrary–precision integers</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Fixed–point numbers</td>
</tr>
<tr>
<td>Channels</td>
<td>C++ user–defined types</td>
</tr>
<tr>
<td>Event–Driven Simulation Kernel</td>
<td></td>
</tr>
</tbody>
</table>

C++ Language Standard
SystemC - AMS concept

Semantic layer
- Semantic 1.1
- Semantic 1.2
- Semantic 2.1

Solver layer
- Solver 1
- Solver 2

Synchronization layer
- AMS - Synchronization

SystemC layer
- SystemC kernel

Classical SystemC Layers
Synchronization Layer

♦ Must support accurate and fast mechanism

♦ Must encapsulate different solvers and solver instances

♦ Must be such generic as possible

♦ Must have a limited complexity

♦ Restrictions has to be defined to achieve the goals
**Solver Layer**

- Providing algorithms for solving equation systems
- Can be high specialized
- Must fulfill the requirements of the synchronization layer
Semantic Layer

♦ Provides the solver with the equation system

♦ Provides the user with an interface

♦ This are netlist description, equation based description, ...
Principle example for the definition of a conservative domain

class sca_lin_elec_prim: public sca_module
{
    virtual void matrix_stamps(); //system of equations contributions
    SCACTOR\{sca_lin_elec_prim\}
        solver->registrate_matrix_stamps(matrix_stamps); }
    sca_lin_elec_solver_if* solver;
};

//implementation of a resistor
class sca_r : public sca_lin_elec_prim
{
    public:
        elec_port a;
        elec_port b;

        double value;
        void matrix_stamps()
        {
            sca_a( a->node(), a->node()) += 1.0/value;
            sca_a( a->node(), b->node()) += -1.0/value;
            sca_a( b->node(), a->node()) += -1.0/value;
            sca_a( b->node(), b->node()) += 1.0/value;
        }
};
Phases of SystemC-AMS Definition and Implementation

♦ Phase 1:
  • Synchronous dataflow synchronization layer
  • Linear constant step width analog solver
  • Dataflow description, Linear networks, Analog behavior models (transfer function, state space, pole zero)

♦ Phase 2:
  • Variable step width synchronization layer
  • Nonlinear DAE solver, ac-solver
  • Equation based description, Nonlinear networks, Nonlinear behavioral models

♦ Phase 3:
  • Freezing synchronization principles
  • Freezing interfaces for further extension to new domains
  • Providing further MoC’s and methodologies e.g. for baseband modeling
Part 3: Application Examples

♦ Electronic example: PLL
  • Basic AMS language constructs
  • Hierarchical example

♦ Automotive example: PWM driver
  • Typical modeling issues in system design

♦ Telecommunication example: xDSL
  • Combination and interaction of different MoCs
PLL Example

\[ v_{pc}(t) = K_{pc} \left( \varphi_{ref} - \varphi_{vco} \right) = K_{pc} \Delta \varphi \]

\[ f_{vco} - f_{c0} = K_{vco} v_{ctrl} \]

\[ \frac{d \varphi_{vco}}{dt} = K_{vco} v_{ctrl} \]
PLL: Phase Comparator

\[ v_{\text{ref}} = V_{\text{ref}} \sin(\omega_{\text{ref}} t + \varphi_{\text{ref}}) \]
\[ v_{\text{vco}} = V_{\text{vco}} \sin(\omega_{\text{ref}} t + \varphi_{\text{vco}}) \]

\[ v_{pc}(t) = \frac{K_{pc} V_{\text{ref}} V_{\text{vco}}}{2} \cos(\varphi_{\text{ref}} - \varphi_{\text{vco}}) = K_m \cos(\Delta \varphi) \]

// phc.h

#include "systemc-ams.h"

SCA_SDF_MODULE(phc) {
    sca_sdf_in<double> in1;
    sca_sdf_in<double> in2;
    sca_sdf_out<double> out;
    double kpc; // gain

    void sig_proc() {
        out.write(kpc*in1.read()*in2.read());
    }

    SCA_CTOR(phc) {} // phc
PLL: Phase Comparator Testbench (1/2)

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

SCA_SDF_MODULE(ref_src) {
    sca_sdf_out<double> out;
    double ampl, freq; // source amplitude and frequency

    void sig_proc() {
        out.write(ampl*sin(2*M_PI*freq*sc_time_stamp().to_seconds()));
    }

    SCA_CTOR(ref_src) {}
};

...SCA_SDF_MODULE(vco_src) {
    sca_sdf_out<double> out;
    double ampl, freq; // source amplitude and frequency
    double dphi;       // phase shift

    void sig_proc() {
        const double DPHI_STEP = 157.1e-3*0.05; // per 1 us*SDF step
        double vout = ampl*sin(2*M_PI*freq*sc_time_stamp().to_seconds() + dphi);
        dphi += DPHI_STEP; dphi = (dphi > M_PI)? M_PI : dphi;
        out.write(vout);
    }

    SCA_CTOR(vco_src) { dphi = -M_PI; }
};
...```
int sc_main(int argc, char* argv[]) {
    
    sca_sdf_signal<double> ref, vco, pco;
    
    sc_set_time_resolution(1.0, SC_NS);
    
    phc i_pc("pc");
    i_pc.in1(ref);
    i_pc.in2(vco);
    i_pc.out(pco);
    i_pc.kpc = 0.66;
    
    ref_src i_ref_src("ref_src");
    i_ref_src.out(ref);
    i_ref_src.out.set_T(sc_time(0.05, SC_US));
    i_ref_src.ampl = 1.0;
    i_ref_src.freq = 1e6;
    
    vco_src i_vco_src("vco_src");
    i_vco_src.out(vco);
    i_vco_src.ampl = 1.0;
    i_vco_src.freq = 1e6;
    
    trace tr_ref("tr_ref"); tr_ref.sin(ref);
    trace tr_vco("tr_vco"); tr_vco.sin(vco);
    trace tr_pco("tr_pco"); tr_pco.sin(pco);
    
    sc_start(41.0, SC_US);
    
    return 0;
}
### PLL: Loop Filter

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

SCA_SDF_MODULE(lp1) {
    sca_sdf_in<double> in;
    sca_sdf_out<double> out;

double fp; // pole frequency
double h0; // DC gain

double tau; // time constant
double outn1; // internal state
double tn1; // t(n-1)

void init() { tau = 1.0/(2.0*M_PI*fp); }

void sig_proc() {
    double tn = sc_time_stamp().to_seconds();
    double dt = tn - tn1;
    outn1 = (outn1*tau + h0*in.read()*dt)/(tau + dt);
    tn1 = tn;
    out.write(outn1);
}

SCA_CTOR(lp1) { outn1 = 0.0; tn1 = 0.0; }
}; // lp1
```

$$H(s) = H_0 \frac{1}{1 + s\tau_1}$$
PLL: Loop Filter Testbench

LP1.FP = 1kHz
LP1.H0 = 1.0
SRC.AMPL = 1.0
SRC.FREQ = 10kHz

i_src.out.set_T(sc_time(0.005, SC_MS));
sc_start(2.0, SC_MS);
#include "systemc-ams.h"

SCA_SDF_MODULE(vco) {

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

  double gain;   // gain
  double kvco;   // sensitivity [Hz/V]
  double fc;     // central frequency [Hz]
  double vfc;    // control voltage to get FC

  double wc;     // central pulsation [rad/s]
  double kvcor;  // sensitivity [rad/(s*V)]

  void init() {
    wc = 2.0*M_PI*fc;
    kvcor = 2.0*M_PI*kvco;
  }

  …

  void sig_proc() {
    double tn = sc_time_stamp().to_seconds();
    double wvco = (wc + kvcor*(in.read() - vfc));
    out.write(gain*sin(wvco*tn));
  }

  …

  SCA_CTOR(vco) {}  
}; // vco

\[
\begin{align*}
\omega_{vco}(t) &= \omega_c + K_{vco} \left[ u_{ctrl}(t) - V_{c0} \right] \\
\varphi_{vco}(t) &= \int \omega_{vco}(t) dt = \omega_c t + K_{vco} \int \left[ u_{ctrl}(t) - V_{c0} \right] dt \\
v_{vco}(t) &= V_{vco} \sin(\varphi_{vco})
\end{align*}
\]
Part 3

PLL: VCO Testbench

VCO.GAIN = 2.5
VCO.KVCO = 10 Hz/V
VCO.FC = 1MHz
VCO.VFC = 0 V

`sc_set_time_resolution(0.01, SC_US);`
`i_src.out.set_T(sc_time(0.01, SC_US));`
`sc_start(14.0, SC_US);`
#include "systemc-ams.h"
#include "../PHC/phc.h"
#include "../LP1/lp1.h"
#include "../VCO/vco.h"

int sc_main(int argc, char* argv[]) {
    sca_sdf_signal<double> ref, pco, lpo, vcoo;
    phc i_phc("phc");
    i_phc.in1(ref);
    i_phc.in2(vcoo);
    i_phc.out(pco);
    i_phc.kpc = 3.72;

    lp1 i_lp1("lp1");
    i_lp1.in(pco);
    i_lp1.out(lpo);
    i_lp1.fp = 112e3; i_lp1.h0 = 1.0;
    ...

    vco i_vco("vco");
    i_vco.in(lpo);
    i_vco.out(vcoo);
    i_vco.out.set_delay(1); // feedback loop!
    i_vco.gain = 1.0; i_vco.kvco = 3e4;
    i_vco.fc = 7e6; i_vco.vfc = 0.0;

    sc_set_time_resolution(0.001, SC_US);
    src_sin src_ref("src_ref");
    src_ref.out(ref);
    src_ref.out.set_T(sc_time(0.001, SC_US));
    src_ref.ampl = 1.0;
    src_ref.freq = 7e6;

    trace tr_ref("tr_ref1"); tr_ref.in(ref);
    trace tr_pco("tr_pco1"); tr_pco.in(pco);
    trace tr_lpo("tr_lpo1"); tr_lpo.in(lpo);
    trace tr_vcoo("tr_vcoo1"); tr_vcoo.in(vcoo);

    sc_start(120, SC_US);

    return 0;
}
PLL Example: Summary

♦ SDF MoC appropriate for modeling continuous-time behavior
  • Provided the sampling frequency is much higher than operating frequencies

♦ Abstract (signal-flow) models

♦ Minimal coding overhead
  • Predefined member functions $\text{init}$, $\text{sig\_proc}$, …

♦ SDF simulation semantics
  • Time step
  • Loop delay
PWM Example

♦ PWM application

♦ Aims of modeling and simulation

♦ A PWM Driver in SystemC-AMS

♦ Demonstration
Part 3

The PWM in Automotive Applications

♦ Automotive applications, general requirements

  • Environment conditions such as temperature, humidity change dramatically

  • High long-term stability and reliability required

  • Fail-safe, self-diagnostics

♦ Purpose of the PWM power driver:

  • Control a voltage (or a current) by **switching** transistors.

  • Compensate changed parameters by control loop.

  • Provide interface that allows checking of parameters
♦ Control a voltage (or a current) by **switching** transistors.

- T1 closed, T2 open
- T1 open, T2 closed
- T1/T2 opened, closed with PWM

♦ Average voltage (current) determined by ratio on / off of the Transistors T1, T2
Control loop reduces impact of drift, etc.

Functional model, executable specification:

Note: This is not (yet) an architecture; we can implement it in many different ways …
The Questions to Modeling and Simulation

♦ A model is never correct „as it is“
  • A model is used to answer questions of the designer to reality
  • The model is only useful, if the question is precise!

♦ Purposes of SystemC-AMS model of PWM:

1. Verification of overall concept, executable specification
   → Functional model

2. Evaluation of different parameters and architectures
   (partitioning A/D/SW, impact of quantization, sampling, drift, etc.)
   → Computation accurate model

3. Virtual prototype for circuit development, software development and
   overall system simulation
   → Interface accurate model
Aim of modeling and simulation at functional level:

- Keep modeling effort low, model only the required functionality
  → Use MoC, which is natural for modeling the functionality
- No „over-specification“, abstraction from implementation
- High simulation performance

Useful MoC:
Timed Static data flow with very high sampling rate
→ mimics continuous-time block diagram

Modeling behavior of components „as easy as possible“:

- Modeling of power driver by transfer function H(s)
- Modeling of pulse generator by discrete process
- Timed SDF models for PI Controller and adder
Model of PI Controller: Overall Structure

```c
sca_sdf_signal<double>
    uc, deviation, Uprog, correction, on_off;

busif bif1("bif1");
    bif1.out(Uprog);
    bif1.out.set_T(sc_time(0.00005,SC_SEC));

diff add1("add1");
    add1.in1(uc); add1.in2(Uprog);
    add1.out(deviation);

sca_s_pi_ctrl ctrl("ctrl");
    ctrl.x(deviation);
    ctrl.y(correction);
    ctrl.k=10.0; ctrl.T=10.0;

sca_spatial load("load");
    load.x(on_off);
    load.y(uc);
    load.add_pole(255, -1/0.05);

pulse_gen_de pulsegen1("pulsegen1");
    pulsegen1.in(correction);
    pulsegen1.out(on_off);

trace_signal uc_dat("uc");
    uc_dat.in(uc);
```

Delay of 1 step
breaks cyclic dependency!
→ Attribute of port
SCA_SDF_MODULE(busif)
{
  sca_sdf_out<double> out;

  // Master/Slave ports for setting
  // the registers via remote method calls from
  // software
  // ... (not yet)

  // Register map
  // ... (confidential)
  sc_uint<XXX> programmed_value;

  void sig_proc()
  {
    out.write(programmed_value);
  }

  SCACTOR(busif)
  {
    programmed_value = 100;
  }
};
The Adder

// Module that computes the deviation from the
// programmed value.

SCA_SDF_MODULE(diff)
{
    sca_sdf_in<double>  in1, in2;
    sca_sdf_out<double> out;

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

    void sig_proc()
    {
        out.write( -in1.read() + in2.read() );
    }

    SCACTOR(diff);
};
// Module which generates a pulse by a discrete event process.
SCA_SDF_MODULE(pulse_gen_de)
{
    sca_sdf_in<double>   in;
    sca_sdf_out<double> out;

    void pulse_generator()
    {
        do {
            double in_lim = in.read();
            if (in_lim > 255.0) in_lim = 255.0;
            if (in_lim < 0.0)   in_lim = 0.0;

            sc_time on_time = 5*sc_time(in_lim, SC_US);
            sc_time off_time = 5*sc_time(255.0-in_lim, SC_US);

            out.write(1); wait(on_time);
            out.write(0); wait(off_time);
        } while (true);
    }

    SC_CTOR(pulse_gen_de)
    {
        SC_THREAD(pulse_generator); }
};
Analog PI Controller

```
// PI controller
SCA_SDF_MODULE(sca_s_pi_ctrl)
{
    sca_sdf_in<double> x;
    sca_sdf_out<double> y;

    void sig_proc()
    {
        sc_time now = simcontext()->time_stamp();
        sc_time t = now - last_change;
        last_change = now;
        state += x.read() * t.to_seconds();
        y.write(k * (T * state + x.read()));
    }

    SCA_CTOR(sca_s_pi_ctrl)
    {
        last_change = sc_time(0, SC_SEC);
        state = 0.0;
    }

    double k, T;

    protected:
        double state;
        sc_time last_change;
};
```

\[ H(s) = k + Ts \]

sig_proc considers different step widths, and implements "analog" behavior.

\(k, T\) are public and must be set before simulation starts.
### Part 3  Modeling the Power Driver by a Transfer Function

\[ H(s) = \sum_{i=0}^{n} \frac{a_i}{1 - s/pole_i} \]

\[ h(t) = \sum_{i=0}^{n} a_i e^{s/pole_i} + \text{const} \]

**SCA_SDF_MODULE(sca_spartial)**

```cpp
SCA_SDF_MODULE(sca_spartial) {
    sca_sdf_in<double> x;
    sca_sdf_out<double> y;

    void sig_proc() {
        double output=0.0;
        sc_time now = simcontext()->time_stamp();
        sc_time t = now - last_change;
        last_change = now;
        for (register unsigned int i=0; i < a.size(); i++)
            state[i] = (exp(t.to_seconds()*pole[i])*(state[i] - a[i]*x.read()) + a[i]*x.read());
        output += state[i].real();
        y.write(output);
    }
}
```

```cpp
void add_pole(const double& a, const complex<double>& pole) {
    this->a.push_back(a);
    this->pole.push_back(pole);
    this->state.push_back(complex<double>(0.0, 0.0));
}
```

**SCA_CTOR(sca_spartial)**

```cpp
SCA_CTOR(sca_spartial) {
    last_change = sc_time(0, SC_SEC);
}
```

**protected:**

```cpp
vector< double > a;
vector< complex<double> > pole, state;
sc_time last_change;
```
Scenarios for tracing a signal...

```cpp
SCA_SDF_MODULE(trace_signal)
{
    sca_sdf_in<double> in;

    ofstream output;

    void sig_proc()
    {
        output << simcontext()->time_stamp().to_seconds() << "\t" << in.read() << endl;
    }

    SCA_CTOR(trace_signal)
    {
        output.open(name(), ios::out);
    }
};
```
Simulation:
sc_start(0.2, SC_SEC);
Aim of modeling and simulation at computation accurate level:

- Keep modeling effort as low as possible, model only behavior of an implementation
- Model architecture by properties of MoC
- Evaluation of parameters, signal processing methods, architectures partitioning Analog / Digital / Software, impact of quantization, sampling, drift, etc.

Useful MoC:
Timed SDF, but with constant step width = clock cycles, 1 delay / block
→ mimics DSP implementation.
Part 3

„Imitating“ Architectures ...

Digital: \( \text{sc\_uint<BW>}, \)

SDF with \( T = \) clock frequency

Analog: \( \text{double}, \)

SDF with high \( T \)
Part 3

“Imitating“ a DSP Realization - Timing

♦ Use discrete-time

- Start computations at the same points in time as in DSP realization
- Scheduling, allocation of time slots

Modeled by delay at output ports!
Example of Computation Accurate Model: Pulse Generator

SCA_SDF_MODULE(pulse_gen_d)
{
    sca_sdf_in< sc_uint<BW> > in;
sca_sdf_out< double > out;

    unsigned cnt, clocks;

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

    void sig_proc()
    {
        cnt ++;
cnt = cnt%256;

        if ( cnt < in.read() )
        { out.write(1.0); }
        else
        { out.write(0.0); }
    }

    SCA_CTOR(pulse_gen_d)
    {
        clocks = 3;
    }
};

For **bit-accurate** simulation …

Delays at out-ports model the total delay for this module

Instead of DE modeling, we mimic use of a counter for computing the pulse width.

Later, we can refine this easily!
Impact of quantization, timing on system properties:

![Graph showing impact of quantization and timing on system properties.](image)
PWM: Interface Accurate Model

♦ Aim of modeling and simulation at interface accurate level:

- Use of model as a virtual prototype

- Allow coupling with models of an implementation

- HERE:
  Support development of software!
  ➔ We model what the software sees:
  Transaction to/from registers via
  - method calls (without drivers)
  - and/or bus transfers (with SW drivers, OS, … )
In order to come to a computation accurate model, controlling signals such as clock, enable, … are required.

The signals are set here. This makes communication and synchronization explicit.
Part 3

Refinement of Pulse Generator (2)

♦ Discrete event model of digital implementation …

Example: adder at RT level

```
SC_MODULE(diff_rt)
{
    sc_in<bool> add_load;

    sc_in< sc_uint<BW> > in1, in2;
    sc_out< sc_uint<BW> > out;

    if ( add_load == ´1´ )
    {
        out.write( -in1.read() + in2.read() );
    }
}
```
A More Detailed Version of the Bus Interface ...

// Transaction level interface, e.g.
//
// sc_inslave<sc_uint<BW> > inline;
// sc_outslave<sc_uint<BW> > outline;
// sc_in<bool> cs_n;

// internal registers and states of businterface:
// sc_signal< sc_uint<BW> > registers[128];
// sc_uint<7> adress;
// sc_uint<10> value_to_send;
// sc_uint<10> value_to_send_next;
// sc_uint<10> value_received;
// sc_uint<7> value_received_adress;

enum { receive_adress, receive_data } state;

void do_receive();
void do_send();

void print_register_dump(); // for debugging SW ...

SC_CTOR(ssio_slave)
{
    value_to_send = 0;
    state = receive_adress;
    SC_SLAVE(do_receive, inline);
    SC_SLAVE(do_send, outline);
}

// Definition of register map, e.g.:

#define IDENTIFICATION registers[1]
#define RESET          registers[2]
#define UC_0           registers[7]
...
#define AD_CHANNEL_1   registers[13]
...
#define AD_CHANNEL_N   registers[29]
...
#define PWM_CONFIG     registers[30]
...
#define COEF_T         registers[38]
#define COEF_K         registers[39]
#define PWM_MODE       registers[40]
SystemC-AMS allows us the modeling of PWM application

- As executable specification (functional model)
- For comparison of different architectures (computation accurate model)
- For virtual prototyping (interface accurate model)

→ concurrent development of software
→ concurrent development of single modules
ADSL - Example

♦ System overview

♦ Designflow

♦ Used proprietary SystemC-AMS extension

♦ Model examples

♦ A testbench concept
ADSL - Transmission System
Languages/ Tools

- C++
- Matlab
- Framemaker
- C
- Embedded SW
- Visual C++
- SystemC / C++
- System integration
- Analog Design
  - Spice, VHDL-AMS
  - Verilog-AMS
- Digital Design
  - VHDL / Verilog
System Overview Voice Codec

- Linear electrical nets (red)
- Linear DAE's (magenta)
- Data flow, frequency domain (blue)
- Pure SystemC blocks (green)

Diagram shows various components such as SLIC, line, subscriber line, hook, PCM codec, and control blocks.
Used SystemC Analog Extensions
Behavioural Models: Embedding Equation Systems

- Subscriber / line
- Protection net
- SLIC
- AFE
- PCM codec

- Hook
- Line voltage

- 32 MHz / 4 MHz
- 4 MHz down to 2 kHz

- Red: Linear electrical
- Magenta: Linear DAE's
- Blue: Data flow
- Green: Pure SystemC

Red: linear electrical
Magenta: linear DAE’s
Blue: data flow
Green: pure SystemC

Behavioural Models: Embedding Equation Systems

- Prefi_ac
- In (sdf signal)
- Out (sdf signal)

- 32 MHz down to 8 kHz
- 4 MHz down to 2 kHz

- Xgain (SysC signal)
Behavourial Models: Embedding Equation Systems

**SDF_MODULE**(prefi_ac)
{
    sdf_inport<double> in;
    sdf_outport<double> out;
    sc2sdf_inport<bool> xgain;

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

    // states
    LTF_ID ltf_id;       // filter id
    fhg_vector<double> A, B; // coeff vector
    fhg_vector<double> S;   // state vector

    void init() {
        // integration step width
        ltf_id.H = in.get_T().get_time_in_sec();
        // 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(A, B, S, ltf_id, 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;
    }
}

\[
H(s) = \frac{1}{1 + \frac{1}{\frac{1}{2\pi f_c} s}}
\]
Behavioural Models: Frequency Domain

- Red: Linear electrical
- Magenta: Linear DAE’s
- Blue: Data flow
- Green: Pure SystemC
**Frequency Domain Simulation**

SDF_MODULE(ac_tx_comb)
{
  sdf_inport<bool> in;
  sdf_outport<sc_int<28> > out;

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

  void ac_domain(double freq,
                 fhg_matrix<complex<double> >& input,
                 fhg_matrix<complex<double> >& output)
  {
    complexd j(0, 1);
    complexd z = exp(2.0*j*M_PI*freq*in.get_T().get_time_in_sec());
    complexd k = 64; //decimation factor
    complexd n = 3; //order of comb filter

    // complex transfer function:
    complexd h = pow((1-pow(z,-k))/(1-1/z), n);

    output(0) = h * input(0);
  }

  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) {
      ...
    }
  }

  SCA_CTOR
  {
    ...
  }
};

\[
H(Z) = \left( \frac{1-Z^{-k}}{1-Z^{-1}} \right)^n \quad Z = e^{j2\pi f/f_s}
\]
Example Test Bench: Results of Frequency Analysis

**Frequency Response (Transfer Function)**

- amplitude in dB
- frequency in Hz
- ac solver
- sdf_fft result

**Output Spectrum**

- amplitude
- frequency in Hz
- complete at input frequencies

Diagram:
- INPUT
- sdf_fft
- OUTPUT
- reference
- probe
- multi_tone
- lpx3
- DEC 8
- lpx2
- DUT
- other blocks
Linear Electrical Networks

subscriber / line
hook
line voltage
protection net
tip
ring
line drv
SLIC
ctrl
w_it 100p
r_it
r_prefi
w_prefi
1k
2k

PCM codec
ac
afe_ac
afe_dc
prefi
dac
pofi
ctrl
kit

w_prefi

red: linear electrical
magenta: linear DAE’s
blue: data flow
green: pure SystemC

32 MHz / 4 MHz
32 MHz down to 8 kHz
Linear Electrical Networks

```
// electrical node
elec_wire w_it;

// converter signal
elecv2sdf w_prefi;

// reference node
elec_gnd gnd;

// signal tracing
sdf_signal<double> kit;

lsdf i_t;
  i_t.a(gnd);  // pos
  i_t.b(w1);   // neg
  i_t.ctl(kit); // current value by signal

R r_it(2e3);
  r_it.a(w_it);
  r_it.b(gnd);

R r_prefi(1e3);
  r_prefi.a(w_prefi);
  r_prefi.b(gnd);

C c_it(100e-9);
  c_it.a(w_it);
  c_it.b(w_prefi);
```

// signal tracing

```
trace tr1(MATLAB, "tr1.dat");

tr1.add(&w_it); // node voltage
tr1.add(&r_it); // current through r_it
```
Netlist input by schematic entry

ELSDF_MODULE(slic_geminax_nl) {
    //Ports
    sdf_import<double> vin;
    elec_port vsp;
    : 
    struct params //parameter 
    {
        double R1;
        :
        params()
        { //default values 
            R1 = 900.0;
            : 
        }
    };
    C *i_C1p; //Instance 
    R *i_R1p;
    : 
    elec_wire __1; //Nodes 
    sdf_signal<double > __5;
    : void architecture();
    slic_geminax_nl(const char *nm, params pa) : ..
    {
        architecture();
    }
    private: 
    params p;
}; 

slic_geminax_nl.h

void slic_geminax_nl::architecture() 
{
    i_C1p = new C(i_C1p);
    i_C1p -> a(__1);
    i_C1p -> b(__2);
    i_C1p->value = pa.C1;
    : 
    slic_nl_fct::params p_i_slic_nl_fctp;
    p_i_slic_nl_fctp.r2 = pa.R2;
    p_i_slic_nl_fctp.vlim = pa.vlim;
    i_slic_nl_fctp = new slic_nl_fct("i_slic_nl_fctp",
                                         p_i_slic_nl_fctp);
    i_slic_nl_fctp -> in_vout(vlin);
    i_slic_nl_fctp -> vout_nl(vnlp);
    i_slic_nl_fctp -> vopvin(__5);
    : 
    i_convn = new elec_to_sdf("i_convn");
    i_convn -> elec_i(__9);
    i_convn -> sdf_o(vlinn);
    : 
    i_R2n = new R(i_R2n);
    i_R2n -> b(von);
    i_R2n -> a(virtual_gnd);
    i_R2n->value = pa.R2;
    : 
    i_Rsdfp = new Rsdf("i_Rsdfp");
    i_Rsdfp -> ctl(__5);
    i_Rsdfp -> b(virtual_gnd);
    i_Rsdfp -> a(GND);
    : 
} 

slic_geminax_nl.cpp
Spice - SystemC-AMS conversion

transformer.h

```c
#include "mixsigc.h"

ELEC_MODULE(transformer)
{
    elec_port RING_DS;
    elec_port RING_LINE;
    elec_port TIP_LINE;
    elec_port TIP_DS;
    struct params
    {
        double R1,L1,C1,E1,F3;
        params()
        {
            R1 = 2.6;
            L1 = 1.0;
            C1 = 21.51e-12;
            E1 = 1.0;
            F3 = 1.05e-3;
        }
    };
    ...
}
```

transformer.cpp

```c
#include "transformer.h"

void transformer::architecture()
{
    transformer(char* name, params pa):elec_elements(Hierarchic),
    p(pa)
    {
        architecture();
    }
private:
    params p;
};
```

automatic converter

spice 2 SystemC_AMS

```
.SUBCKT transformer RING_D RING_DS TIP_D TIP_DS
C_p $N_0001 $N_0002 10n
C_s $N_0003 $N_0004 10n
R_p $N_0005 RING_D 2260
R_s 0 $N_0002 300
R_32 $N_0004 0 300
R_22 TIP_D $N_0006 2260
R_11 RING_DS RING_D 4.7
.ENDS transformer
```
Mixed-Signal HW-SW - Co-Simulation

32 MHz down to 8 kHz

PCM codec

control

dc

32 MHz down to 2 kHz

Mixed-Signal HW-SW - Co-Simulation

32 MHz down to 8 kHz

PCM codec

control

if(M1 & M2 & M3) active_mode = 1;if(active_mode & !active_last) start_load=1;

if(active_mode)
{
    switch_on_dsp=1;switch_on_analog_bias=1;
}

if(!hook_detected&active_mode & !detection_suppresed)

hook_level=active_low_level;

switch(slic_mode)
{
    case SLIC_LOAD_MODE:
        c1l= load & !hook_detected & last_active;

        break;

}
A Testbench concept
STIMULI(example_stimuli), vinetic_ctrl_utilities
{

STIMULICTOR(example_stimuli)
{
    //if name not assigned name equals to the stimuli name (example_stimuli)
    //the name can be used for stimuli selection
    name = "ex";

    short_description = "example stimuli";

    description =
    "This stimuli demonstrates the use of the stimuli base class \n"
    "The CVS status of this stimuli is : $Id: example_stimuli.h,v 1.3 2004/06/14 17:27:57 markwirt Exp $\n";
}

//this methods can be optional provided
void tb_configuration(); //set tb parameters
void tb_measure_inclusion(); //called after dut instantiation
void traces(); //called before simulation start
void stimuli_sequence(); //stimuli sequence
void simulation_control(); //simulation control commands

};

element_stimuli.h
void example_stimuli::tb_configuration()  //set dut parameter
{
    //assign DUT parameters
    dut_p->line_fpath = "../../../geminax_a/DEBUG/line_files/short_cut.systemc";
}

void example_stimuli::tb_measure_inclusion()  //instantiate additional modules for stimulation and post-processing
{
    connect_file_in(dut->i_vinetic->s_pcm_in,"pack2cod_rx.dat");
    connect_file_out(dut->i_vinetic->i_vinetic_lm->pack2cod_rx,"pack2cod_rx_out.dat");
}

void example_stimuli::traces()  //define signals for tracing
{
    sca_trace *tf = new sca_trace(MATLAB, "ivd.dat");
    tf->add(&(dut->w_ring_vinetic_geminax) ,"w_ring_vinetic_geminax");
}

void example_stimuli::stimuli_sequence()  //dynamic stimuli sequence
{
    wait(1.0,SC_MS);
    set_sopi(mode_src, PDR);  //change system mode
    dut->hook = 1;                //set subscriber to offhook
    wait(1000.0,SC_MS); sc_stop();
}

void example_stimuli::simulation_control()  //simulation control commands
{
    sc_start(-1);
}

example_stimuli.cpp
Stimuli registration

```c
#include "example_stimuli.h"
#include "geminax_a0_stimuli.h"
#include "dcchar_test_stimuli.h"
#include "ring_ext_test_stimuli.h"
#include "amazon_test_stimuli.h"

void registrate_stimulis()
{
    avail_stimulis.push_back(new example_stimuli);
    avail_stimulis.push_back(new geminax_a0_stimuli);
    avail_stimulis.push_back(new dcchar_test_stimuli);
    avail_stimulis.push_back(new ring_ext_test_stimuli);
    avail_stimulis.push_back(new amazon_test_stimuli);
}
```

stimulis.h
The Testbench

```cpp
int sc_main(int argc, char** argv)
{
    registrate_stimulis();
    stimuli_num = stimuli_selector(argc, argv, avail_stimulis, default_stimuli);

    stimuli_base* stimuli = avail_stimulis[stimuli_num];
    cout << "Start Stimuli:" << endl;  stimuli->print_id_shortdescr(stimuli_num);
    stimuli->tb_configuration();
    ivd_toplevel* dut = new ivd_toplevel("dut", tp);   // dut instantiation
    stimuli->set_dut(dut);stimuli->tb_measure_inclusion();mixsigc_init();
    stimuli->traces();
    stimuli->simulation_control();
    cout << endl << sc_time_stamp() << " Simulation finished" << endl << endl;
    mixsigc_finish();
}
```

`ivd_testbench.cpp`
Example Simulation
Example Simulations
Conclusions

♦ Emerging applications are more and more heterogeneous in nature (digital & analog HW, SW, non electrical parts)

♦ System design requires increasingly efficient verification means:
  • Efficient modeling means (variety of interacting MoCs, abstraction)
  • Efficient simulation means (optimized simulation kernels)

♦ SystemC is providing an appropriate environment for supporting these requirements

♦ SystemC-AMS targets the same objectives for mixed discrete/continuous systems
  • Prototype based on SDF MoC
  • More information on http://www.ti.informatik.uni-frankfurt.de/systemc-ams